UNIVERSITÉ DU LUXEMBOURG

PhD-FSTC-2015-48
The Faculty of Science, Technology and Communication

Dissertation

Defence held on the 9th of October 2015 in Luxembourg

to obtain the degree of

Docteur de l'Université du Luxembourg en Informatique

by

Kevin ALLIX
Born on the 21st of October 1982 in Saint-Lô (Manche, France)

Challenges and Outlook in Machine Learning-based Malware Detection for Android

Dissertation Defence Committee

Dr.-ing Yves Le Traon, Dissertation Supervisor
Professor, University of Luxembourg, Luxembourg

Dr. Jacques Klein, Chairman
Senior Research Scientist, University of Luxembourg, Luxembourg

Dr. Tegawendé F. Bissyandé, Vice Chairman
Research Associate, University of Luxembourg, Luxembourg

Dr. Lorenzo Cavallaro
Senior Lecturer, Royal Holloway – University of London, U.K.

Dr. Christian Rossow
Research Group Leader, Saarland University, Germany

Abstract

Just like in traditional desktop computing, one of the major security issues in mobile computing lies in malicious software. Several recent studies have shown that Android, as today's most widespread Operating System, is the target of most of the new families of malware.

Manually analysing an Android application to determine whether it is malicious or not is a time-consuming process. Furthermore, because of the complexity of analysing an application, this task can only be conducted by highly-skilled—hence hard to come by—professionals.

Researchers *naturally* sought to transfer this process from humans to computers to lower the cost of detecting malware. Machine-Learning techniques, looking at patterns amongst known malware and inferring models of what discriminates malware from goodware, have long been summoned to build malware detectors.

The vast quantity of data involved in malware detection, added to the fact that we do not know *a priori* how to express in technical terms the difference between malware and goodware, indeed makes the malware detection question a seemingly textbook example of a possible Machine-Learning application.

Despite the vast amount of literature published on the topic of detecting malware with machine-learning, malware detection is not a solved problem. In this Thesis, we investigate issues that affect performance evaluation and that thus may render current machine learning-based malware detectors for Android hardly usable in practical settings, and we propose an approach to overcome those issues. While the experiments presented in this thesis all rely on feature-sets obtained through lightweight static analysis, several of our findings could apply equally to all Machine Learning-based malware detection approaches.

In the first part of this thesis, background information on machine-learning and on malware detection is provided, and the related work is described. A snapshot of the malware landscape in Android application markets is then presented.

The second part discusses three pitfalls hindering the evaluation of malware detectors. We show with extensive experiments how validation methodology, History-unaware dataset construction and the choice of a ground truth can heavily interfere with the performance results of malware detectors.

In a third part, we present an practical approach to detect Android Malware in real-world settings. We then propose several research paths to get closer to our long term goal of building practical, dependable and predictable Android Malware detectors.

Acknowledgements

In May 2011, my good friend Erwan told me about a professor in Luxembourg who might be interested in having me doing a PhD in his team. Just a few days later, I received an e-mail from Prof. Yves Le Traon requesting more information about me, and asking a simple question: "Would you be ready to do a PhD in Luxembourg?"

As readers might guess, I replied I was ready. It seems that I also managed to convince Yves that I was somehow ready.

Four years later, it is now clear that I did not have the faintest idea of where the journey I had embarked on would lead me. "It's the journey that counts, not the destination" as the saying goes, and fortunately, I was not alone during my trip. Several people had a profound impact not only on my work, but also supported me in many ways, allowing me to end up in a destination I am happy with. To them I want to express my gratitude.

I want to take this opportunity to thank Yves Le Traon for accepting me in his team, and for allowing me to follow my interests. He also let me spend time on risky research, but managed to keep on reasonable tracks.

I am grateful to Jacques Klein, who has been my day-to-day guide throughout the sometimes daunting world of academia. While working with me, he demonstrated levels of patience that were unheard of.

My sincere thanks also go to Tegawendé Bissyandé. Our countless discussions on experiment designs and on possible interpretations of experimental results helped me transform many vague insights into deep understanding.

I would like to give special thanks to Quentin Jérôme who first got me interested in Android Malware Detection with Machine Learning. Our collaboration, while too short, has been very fruitful in terms of research, and particularly enjoyable.

Finally, I address my deepest thanks to my very good friends Soliman, Guillaume, Frédéric and Erwan for their continued support. The role they played in this thesis is more important than they can possibly know.

Kevin Allix

Luxembourg, August 2015

Contents

List of Figures

List of Tables

Chapter 1

General Introduction

Contents

1.1 The Rise of the Smartphone...

While modern smartphones started to be popular around year 2005, most notably with BlackBerry devices, the smartphone market really took up just a few years later when Apple released its iPhone in 2007 followed in 2008 by a phone built by HTC and running an operating system developed by Google and named Android.

In 2014 alone one billion smartphones running Android were sold[1]. In not even a decade, smartphones went from being expensive gadgets for either tech-savvy early adopters or business executives to ubiquitous devices now owned by all categories of peoples.

From teens to pensioners, in rich nations like in developing countries, for *geeks* and for persons not interested in technology, the word *phone* is indeed becoming a synonym of *smartphone*.

1.2 ...And of various Smart Devices

Making everything smart and connected to the Internet is nowadays a strong trend. Manufacturers who want to integrate to their products the new functionalities made possible by embedding a networked computer are facing the choice of either designing their own hardware platform and developing all the required software, or of using standard hardware running a standard software stack.

Android is quickly becoming the *de facto* standard for embedding a full-fledged computer into any kind of objects. Partly, this could be explained by the fact that Android runs on the three most prevalent CPU architectures that are x86, ARM and MIPS, in their 32 bits versions like in their 64 bits versions. An element that contributed probably even more to Android's adoption in embedded systems is to be found in Android licensing scheme. By releasing most of Android under an Open Source license, Google allowed any company to use Android on its products without paying anything to Google, unlike traditional, proprietary software stacks dedicated to embedded computing. Finally, success bringing even more success, using Android offers the possibility to also use the huge library of Android applications already developed.

Combined with the availability of many ultra-cheap ARM System-on-Chip solutions, this led to a situation where even the most mundane objects may now integrate an Android-powered computer and add a 'smart' to their name. Examples include smart TVs, Set-Top-Boxes and smart watches. But Android also made its way up to objects few peoples would have envisioned any

[1] http://www.cnet.com/news/android-shipments-exceed-1-billion-for-first-time-in-2014/

reason to connect to the Internet just a decade ago. Besides in-car infotainment systems[2] or cameras, Android is also present in kitchens with smart refrigerators[3]and ovens[4]. Even niche markets such as aquarium controllers[5] can now become 'smart'.

Additionally, smartphones being ubiquitous, they are also used to control other devices that do not run Android. Examples of remotely-controllable devices include thermostats[6], washing machines[7] and toilets[8].

1.3 A Target of Choice

The increasing adoption of smartphones and electronic tablets has created unprecedented opportunities of damages by malicious software which are hidden among the millions of mobile apps available, often for free, on application markets (Felt, Finifter, Chin, Hanna, & Wagner, 2011). This reality is currently witnessed on the Android platform, where more and more users of Android-enabled smartphones and other handheld devices are able to install third party applications from both official and alternative markets. In such a context, the security of devices as well as the security of the underlying network have become an essential challenge for both the end users and their service providers.

Malware pose various threats that range from simple user tracking and leakage of personal information (Enck, Octeau, McDaniel, & Chaudhuri, 2011), to unwarranted premium-rate subscription of SMS services, advanced fraud, and even damaging participation to botnets (Pieterse & Olivier, 2012).

Because smartphones are used to manage nearly all aspects of modern life, each smartphone is a gold-mine of personal information such as location history, web-browsing habits, frequency and content of discussions. Additionally, it is easy for an attacker to monetise a compromised smartphone. With their connections to the phone network, they can place calls and send SMS which enable attackers to earn money without having to enter the muddy waters of personal information black-markets, or to blackmail devices' owner.

Another side-effect of the success of smart-devices in general, and of Android in particular, is that the field of IT Security considerably expanded. Not so long ago, security researchers had nothing

[2] http://appleinsider.com/articles/14/01/07/audi-shows-off-smart-display-android-powered-in-car-tablet
[3] http://gadgets.ndtv.com/others/news/samsungs-t9000-smart-refrigerator-runs-on-android-includes-apps-like-evernote-and-epicurious-320610
[4] http://www.androidcentral.com/hands-second-gen-android-oven
[5] http://www.androidcentral.com/vertex-cereba-froyo-powered-aquarium-controller
[6] https://nest.com/thermostat/meet-nest-thermostat/
[7] http://www.androidcentral.com/socks-samsungs-washing-machine-android-app
[8] http://www.huffingtonpost.com/2012/12/18/satis-bluetooth-toilet-japan-lixil-smartphone_n_2322012.html

to say about kitchens' appliances. Nowadays, even toilets are subject of security vulnerability reports[9].

1.4 Detecting Malware

Although the threat of being infected with a malware is equally important in both the desktop computing world and the mobile computing world, most users of handheld devices fail to realise the severity of the dangers these devices expose them to. This situation is further exacerbated by the fact that antivirus vendors have not yet achieved the same kind of performance that they have achieved for personal computers, nor will they be given the time to do so by developers of mobile malware.

The momentum of malware detection research is growing, stimulated by the rapid spread of mobile malware. Indeed, given the large number of mobile applications being created, combined with the cost of manually analysing those applications, the need for automated malware detection is urgent.

Machine-Learning has often been investigated as a promising tool to help detecting malware at a low cost.

1.5 This Thesis

Many challenges remain before a reliable Android malware detector based on Machine-Learning techniques is demonstrated to perform well in practice. In this thesis, we identify and investigate several of these challenges, and we propose solutions with the aim of building dependable Machine Learning-based Android Malware Detectors.

In **Part I**, we provide background information on machine-learning and on its application to the malware detection problem (chapter 2), and we present the research already done in that field (chapter 3).

We then proceed in **Part II** to explore the world of Android Application with the goal of acquiring a snapshot of the malware landscape in Android application markets.

[9] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-4866

Part III investigates and discusses three pitfalls hindering the evaluation of malware detectors. We show with extensive experiments how validation methodology (chapter 6), History-unaware dataset construction (chapter 7) and the choice of a ground truth (chapter 8) can heavily interfere with the performance results of malware detectors.

Finally, we present in **Part IV** an approach to build a practical and dependable Android Malware detector, and we discuss possible research paths towards building malware detectors that are highly dependable and usable in realistic settings. We conclude this thesis in chapter 11.

Part I

Background

Chapter 2

Background on Machine-Learning and Malware Detection

Contents

2.1 Machine-Learning

Machine-Learning is the field of study and application of algorithms that seek patterns in data. As most Artificial Intelligence sub-domains, Machine-Learning is commonly used in classification problems i.e., where the problem is about determining of which *class* is a given object. In the context of Machine-Learning, a *class* represents a trait, and is domain-specific.

Another possible usage of Machine-Learning techniques deals with Regression: Instead of associating an object with one or several classes (categorical values), objects can be associated with a real value representing either a physical property of the object—like age, height or weight—or any other variable—for example, a student's GPA, or the probability of being green-eyed—we think may be correlated, directly or indirectly, with measurable characteristics of the object.

A vast quantity of problems in many domains *are*—or can be transformed into—classification or regression problems. Random examples of such problems where Machine-Learning techniques are used are the automatic detection of oil spills with Satellite images (Kubat, Holte, & Matwin, 1998; Cococcioni, Corucci, Masini, & Nardelli, 2012), detecting prostate cancer (Madabhushi, Shi, Feldman, Rosen, & Tomaszewski, 2006), forecasting river water quality for agriculture (Liu & Lu, 2014), detecting drowsy drivers (Vural et al., 2009), or forecasting trend reversals in financial markets (Azzini, De Felice, & Tettamanzi, 2012).

2.1.1 General Process

Two main families of Machine-Learning techniques exist:

Supervised Learning Where we have to feed the learning algorithm with a reference dataset containing the class of interest. In supervised learning, the goal is to *teach* a machine to predict the class of unseen objects.

Unsupervised Learning where there is no reference classification, and no specific class of interest. In Unsupervised learning, the goal is often to find groups, or clusters, of objects that are *similar*.

While unsupervised learning can also be useful in the context of malware detection, we focus in this thesis on supervised techniques.

2.1.1.1 Feature Set

Computers in general, and machine-learning algorithms in particular, have no *understanding* of the objects they work on. More precisely, computers work on data that we humans can mentally map to real-world objects, but not on objects themselves.

A common first step to any machine-learning use is to devise a feature set, a set of characteristics of objects that will be used in place of objects.

While one of the many machine-learning functions is to detect correlations that we did not know existed, practitioners usually try their best to include in the feature set characteristics they know—or assume—correlate with the variable of interest.

For very large and/or very complex objects, such as Android applications, the number of possible characteristics, and *a fortiori* the number of possible sets of characteristics is virtually infinite.

As noted in chapter 3, many different families of features have already been tried for Android malware detection, including characteristics obtained through advanced or lightweight static analysis, through dynamic analysis, or features related to meta-data such as permissions or assets.

In this thesis, we describe two of the three different feature sets we used in our experiments. The first one, using a basic-block textual representation is detailed in chapter 6. Next our feature set based on code metrics is introduced in chapter 8. Finally, our feature set that uses sequences of Opcodes is described in (Jerome, Allix, State, & Engel, 2014).

Finding the right feature set that works well for the right application often is the key to successful Machine-Learning. Moreover, a feature set that brings excellent results for finding buggy applications may very well be largely incapable of discriminating malware from goodware. Thus, feature sets are often built on insights. For example, our feature set based on opcode sequences is the logical outcome of the insight that some sequences of instruction might be more likely to be found in malware than in goodware.

2.1.1.2 Features Extraction

Once the choice of features is made, it is necessary to extract those features to build Feature Vectors. This step can be seen as a function whose input is one *object* (in the case of interest here: an Android application) and that outputs an ordered sequence of characteristics. After being run on each application, it is possible to build a feature matrix, i.e., a list of all the feature vectors. Table 2.1 depicts a feature matrix, where, as is common in machine learning, two columns have been added:

Table 2.1: Feature Matrix Representation

	Feature_1	F_2	F_3	⋯	F_n	Class
App_1	True	4	0.4	⋯	SPORTS	Malware
App_2	True	1	1.3	⋯	GAME	Goodware
⋮	⋮	⋮	⋮	⋱	⋮	⋮
App_m	False	12	0.01	⋯	PRODUCTIVITY	Goodware

A sample Identifier so that is is possible to know what application is represented by a feature vector. In our example table, an identifier is in the first column.

A Class taken from a reference classification when available. In our example, the class is either Malware or Goodware.

Table 2.1 also shows that features can be of different types, such as boolean, integer, real, or categorical.

2.1.1.3 Learning

Once a feature matrix is built, it can be fed to a machine-learning algorithm that will try to build a model of what discriminates goodware from malware. This phase is the learning part of the process, where the algorithm tries to derive rules from the example data it has access to, in a way similar—it is hoped—to what makes us humans able to tell a tree from a car. We have seen many *things* that were called either tree or car, and we managed to *generalise* the *concept* of what is a tree and what is a car so that we can identify a car even at our first encounter with this specific model of car.

A feature matrix that is used for training a machine-learning algorithm is called a train set.

A great many learning algorithms exist. In this thesis we use up to four popular algorithms, taken from different families of Machine-Learning algorithms: Support Vector Machine (SVM), the RandomForest ensemble decision-trees algorithm, the RIPPER rule-learning algorithm and the tree-based *C4.5* algorithm.

2.1.1.4 Applying the Model

The model built during learning can now be applied to a set of applications. During this phase, the output is a class prediction for each application of the Testing set. The prediction can either be directly a class name, or can be a real number, usually representing a likelihood of belonging to one class.

If the testing set also comes with a reference classification, or ground truth, it can be compared to the predictions made allowing performance metrics to be computed.

2.2 Metrics

Several Metrics can be used to describe the performance characteristics of a classifier. We present here various metrics that will be used throughout this thesis. While those metrics are relevant to any classification problem, we introduce them with malware detection in mind, and hence use vocabulary specific to malware detection instead of generic, application agnostic terms often found in machine learning literature.

A good understanding of these metrics, and of their limitations, is necessary to fully comprehend the meaning and the scope of the performance results provided in this thesis, as well as in the rest of machine learning-based malware detection. As noted by Bowes, Hall, and Gray (2014), although for a different field of research, Machine-Learning practitioners can take a liberal approach to the choice of metrics they report, making it hard, or plainly impossible, to compare results from an approach to another.

2.2.1 Precision, Recall & F-Measure

Precision, as captured by equation 2.1, quantifies the effectiveness of the tool to identify suspicious applications that are actually malware. When the tool reports applications as malware and all turn out to be as such, its Precision amounts to 1.

$$Precision = \frac{|\{labeled\ malware\} \cap \{malware\ inferred\ by\ tool\}|}{|\{malware\ inferred\ by\ tool\}|} \tag{2.1}$$

Recall on the other hand explores the capability of the tool to identify most of the malware. Equation 2.2 provides the formula for its computation. A Recall evaluated to 0 indicates that no actual malware in the test set has been identified as such by the tool.

$$Recall = \frac{|\{labeled\ malware\} \cap \{malware\ inferred\ by\ tool\}|}{|\{labeled\ malware\}|} \tag{2.2}$$

F-Measure is the harmonic mean between Recall and Precision. We consider that both Precision and Recall are equally important and thus, they are equally weighted in the computation of F-measure in Equation 2.3.

$$\textit{F-Measure} = F_1 = 2 \cdot \frac{Precision \times Recall}{Precision + Recall} \tag{2.3}$$

2.2.2 ROC

Precision, recall and F_Measure are all scalar values. However, it is often interesting to know how an approach performs not only for one set or parameters, but also for a continuum of parameters. In particular, Machine Learning classifiers can often be tuned to either err on the side of caution, or to instead be more aggressive. In the context of malware detection, some will seek to build with a classifier that catches every single potential malware at the cost of benign apps being mis-reported as malicious. Alternatively, the goal may be to catch as many malware as possible, but never mis-classify a benign app. One scalar value cannot represent the performance of a classifier all over the course of the safe/agressive slider. Instead, we often present this performance as a ROC (Receiver Operating Characteristic) Curve. A ROC curve plots the True Positive Rate (TPR, also named Recall) on the Y-axis against the False Positive Rate (FPR, on the X-axis).

2.2.3 AUC: Area Under the ROC Curve

While not as informing as the full ROC curve, the AUC (for Area Under a ROC Curve) is a simple way to obtain one real number that summarise the overall behaviour of a classifier. Given that AUC is a aggregate value, it can be hard to interpret. In particular, comparing AUC values for ROC Curves that have different shapes can lead to erroneous conclusions.

2.3 Malware Detection Specificities

While Machine-Learning techniques can be applied in a wide variety of fields, the Malware detection problem carries several specific issues of its own. Those issues must be kept in mind since they may impose restrictions on what can be done with machine learning.

2.3.1 Scarcity of Ground Truth

There is a common problem faced by all researchers working in the malware detection community: There is no comprehensive and reliable Ground Truth.

While some binaries are known with a very high certainty to be malware, such reliable knowledge is very rare. On the several millions of Android applications packages in existence, only a few hundreds have been demonstrated in published work by an expert to be a malware. Even harder to come by are documented reports of analyses establishing a given application is not a malware.

This fact is one of the reasons why building automated malware detectors is a goal of many research teams in the world, but at the same time, it also poses issues to the very same teams when evaluating their malware detectors.

Also, manually analysing a software is a long process that requires highly skilled professionals, and hence is often not a possibility, especially for large scale studies.

A whole industry, making more than $4 billions a year[1] exists around this very problem. Antivirus vendors claim they know which application is a malware and which is not. However, as is discussed in depth in chapter 8, the consensus among the different antivirus products is extremely limited, making the process of building a reference classification based on knowledge from those claiming to know a difficult task. Even more importantly, such a reference classification would not be universally accepted, and hence would hardly deserve the name of Ground Truth.

[1] http://www.theregister.co.uk/2006/06/26/av_market_gartner/

2.3.2 Precision Vs. Recall Trade-off

In the context of malware detection, not all errors are equally bad. For example, in an hypothetical super-high security environment, not detecting and therefore not blocking a malware may have devastating consequences, while mistakenly blocking a few non-malware may be seen as an acceptable cost. In other environments though, having a few malware might not be perceived as a highly critical event, but being mistakenly prevented from launching a non-malware software—say a spreadsheet—could have unacceptable financial consequences. In such different environments, even the definition of what constitutes a malicious action may very well be different.

This simple example shows that there is no—there cannot be—such a thing as a *perfect* antivirus product: The expectations on what an antivirus products *should* do are not universally shared.

Often, and in particular with Machine Learning-based approaches, a trade-off has to be made between precision and recall. In other words, willing to catch every single malware increases the risk of mistakenly reporting a benign software as a malware. Symmetrically, willing to never block an application that should not be blocked comes with an increased risk of letting actual malware through.

Chapter 3

Related Work

Contents

In this chapter, we enumerate a number of related works to emphasise the importance of understanding the development of malware in order to devise efficient techniques for their detection. These related works span from empirical studies on datasets of malware to malware detection schemes.

3.1 Malicious datasets analysis

Researchers have already shown interest in malicious application datasets analysis. Felt et al. (2011) have analyzed several instances of malware deployed on various mobile platforms such as iOS, Android and Symbian. They detail the wide range of incentives for malware writing, such as users' personal information and credentials exfiltration, ransom attack and the easiest way to profit from smartphone malware, premium-rate SMS services. Their study is however qualitative, while we have focused in chapter 5 on a quantitative study to draw generalisable findings on common patterns.

The Genome dataset, a source of well-established malware, was built as part of a study by Zhou and Jiang (2012). They expose in details features and incentives of the current malware threat on Android. They also suggest that existing antivirus software still need improvements. Our analysis also comes to this conclusion when we demonstrate that most malware cannot be found by all antivirus products.

Opposite to the lightweight forensic analysis approach we leveraged in chapter 5, Enck et al. (2011) performed an in-depth analysis of Android applications by using advanced static analysis methods. Doing so, they were able to discover some risky features of the android framework that could be used by malicious applications. However, our approach allowed to highlight interesting patterns that are could be leveraged more easily.

3.1.1 Dynamic analysis

Various solutions have been proposed to detect malicious Android applications. Crowdroid, presented by Burguera, Zurutuza, and Nadjm-Tehrani (2011), performs dynamic analysis of Android applications by first collecting system calls patterns of applications, and then applying clustering algorithms to discriminate benign from suspicious behaviours. Crowdroid strongly rely on crowd sourcing for system calls patterns collection.

Vidas and Christin (2013) has investigated applications from alternative markets and compared them to applications from the official market. They have found that certain alternative markets

almost exclusively distribute repackaged applications containing malware. They have proceeded to propose AppIntegrity to strengthen the authentication properties offered in application marketplaces. Our findings presented in chapter 5 are in line with theirs, when we note that malware seem to be mass produced, and that the same certificates overlap between malware and benign applications.

3.1.2 Similarity and Heuristics based malware detection

In order to detect repackaged applications—an operation often performed by malware authors to embed their malicious payloads— Zhou, Zhou, Jiang, and Ning (2012) presented *DroidMOSS*. Their approach consists in building a signature of the whole application by using a fuzzy hashing technique on the application's opcodes. Then a similarity score is computed for all Apps of a reference dataset, thus concluding to the detection of a repackaged application if the similarity score is higher than a given threshold

DroidRanger presented by Zhou, Wang, Zhou, and Jiang (2012) tries to detect suspicious applications by first performing a fast filtering step based on permissions requested by an application. It then analyse the application code structure, as well as other properties of applications. Finally, an heuristics-based detection engine is run with the data gathered about applications. With this approach, the authors were able to find malware on the official Android market but also two zero-day malware.

Regarding information leakage detection, Zhou, Zhang, Jiang, and Freeh, 2011 also proposed TISSA, allowing end-users to have a fine grained control of the access to their personal data.

3.1.3 On device mitigation

The topic of embedded mitigation solution was covered by a wide range of previous works.

XManDroid (Bugiel, Davi, Dmitrienko, Fischer, & Sadeghi, 2011) provides a mechanism capable of analyzing Inter Process Communications and decide if connections between applications are compliant with the system policy. This full dynamic solution addresses the problem of application level privilege escalation introduced in (Davi, Dmitrienko, Sadeghi, & Winandy, 2011).

DroidChecker (Chan, Hui, & Yiu, 2012) attends to address the same issue by tracking permissions from the manifest files until their utilization within the application. To achieve this, Chan et al. proposed the use of control flow graphs and taint checking techniques.

Apex (Nauman, Khan, & Zhang, 2010) proposes an extension to the Android permission manager allowing users to customize permissions owned by applications.

Kirin (Enck, Ongtang, & McDaniel, 2009) extends the package installer and analyze permissions before installation. It embeds security rules based on permissions sets and can prevent a program from being installed according to the permissions it requests. A similar approach has been presented in (Bartel, Klein, Monperrus, Allix, & Le Traon, 2012).

Enck et al. introduced Taintdroid (Enck et al., 2010) which uses taint analysis techniques to detect sensitive data leaks and warn the end-user by showing him/her with relevant information.

Hornyack, Han, Jung, Schechter, and Wetherall, 2011 have also studied data leaks. Hornyack et al. present a framework capable of shadowing sensitive user data and of blocking outgoing connections implying data leaked.

3.1.4 Miscellaneous approaches

An offensive framework was presented by Höbarth and Mayrhofer, 2011 which embeds a broad range of available Android exploits such as *Rage Against The Cage*, known to overflow the number of processes allowed. In the wild, this exploit is used by various malware. The framework is able to run arbitrary root exploit and to maintain privileges among reboots.

Rootkits possibilities on smartphones are exposed in (Bickford, O'Hare, Baliga, Ganapathy, & Iftode, 2010), showing that smartphones are as vulnerable as desktop computers. The most valuable incentive to deploy rootkits on smartphones would be the interesting personal data such as voice communications and location.

With *Androguard*[1], Desnos et al. provide a tool to decompile Android applications and perform code analyses (Desnos, 2012). Building on top of these features, *Androguard* also provides a way to detect a large selection of malware, and to measure the similarity of two applications, to detect repackaging for instance.

Finally, concerning the detection of private data leaks, static analysis tools (Octeau et al., 2013; Bartel, Klein, Monperrus, & Le Traon, 2012), including taint analysis (Arzt et al., 2014), have been proposed to deal with the specificities of Android.

[1] http://code.google.com/p/androguard/

3.2 Machine Learning-based Malware Detection

A significant amount of Machine Learning approaches to malware detection has been presented to the research community. Although most of those approaches could not be reproduced due to undisclosed parameters and/or undisclosed datasets, we try to compare their evaluation metrics with our most-closely *in the lab* classifiers presented in chapter 6. None of the approaches introduced by the literature discussed in this section provide a large scale evaluation of their approach.

3.2.1 Android malware detection

In 2012, Sahs and Khan (2012) built an Android malware detector with features based on a combination of Android-specific permissions and a Control-Flow Graph representation. Their classifier was tested with k-Fold [2] cross validation on a dataset of 91 malware and 2 081 goodware. We obtained comparable values of recall but much higher values for precision and F-measure.

Using permissions and API calls as features, Wu, Mao, Wei, Lee, and Wu (2012) performed their experiments on a dataset of 1 500 goodware and 238 malware. Many of our classifiers exhibit higher values of both precision and recall than theirs.

In 2013, Amos, Turner, and White (2013) leveraged dynamic application profiling in their malware detector. The evaluation metrics of their 10-Fold experiment are slightly lowers than ours.

Demme et al. (2013) also used dynamic application analysis to perform malware detection with a dataset of 210 goodware and 503 malware. Many of our *in the lab* classifiers achieved higher performance than their best classifier.

Yerima, Sezer, McWilliams, and Muttik (2013) built malware classifiers based on API calls, external program execution and permissions. Their dataset consists in 1 000 goodware and 1 000 malware. Many of our *in the lab* classifiers achieved higher performance than their best classifier.

Canfora, Mercaldo, and Visaggio (2013) experimented feature sets based on SysCalls and permissions. Their classifiers, evaluated on a dataset of 200 goodware and 200 malware, yielded lower precision and lower recall than ours.

[2] The value of k used by Sahs & Khan was not disclosed.

3.2.2 Windows malware detection

Kolter and Maloof (2006) performed malware classification on Windows Executable files. Using n-grams extracted from those binary files, and the Information Gain feature selection method, they obtained high performance metrics with 10-Fold experimentations on two collections: The first one consisting in 476 malwares and 561 goodware, the second one containing 1 651 malware and 1 971 goodware. Many of our *in the lab* classifiers achieved higher performance metrics.

In 2006, Henchiri and Japkowicz (2006) provided experimental results of a malware detector based on a sophisticated n-grams selection algorithm. They evaluated their classifier using 5-Fold[3] on a dataset of 3 000 samples, of which 1 512 were malware and 1488 were goodware. The majority of our classifiers achieved better results than Henchiri & Japkowicz best ones, even though we used a simple feature selection method.

Zhang, Yin, Hao, Zhang, and Wang (2007) leveraged a multi-classifier combination to build a malware detector. They evaluated the quality of their detector with the 5-Fold method on three datasets, each containing 150 malware and 423 goodware. The features they are using are based on n-grams, and are selected with InfoGain. Zhang et al. mentions testing on a larger dataset as a future work.

Schultz, Eskin, Zadok, and Stolfo (2001) performed malware detection using strings and byte sequences as features. They obtained very high recall and precision with 5-Fold Cross Validation on a dataset of 4 266 Windows executables (3 265 known malicious binaries and 1 001 benign). Many of our classifiers performed similarly good or better.

Perdisci, Lanzi, and Lee (2008b) built a packed executable detector that achieved near 99% accuracy. Their classifiers were trained on 4 493 labelled executables and then tested on 1 005 binaries. The same authors leveraged their packed executable detection method (Perdisci, Lanzi, & Lee, 2008a) and added two malicious code detectors, one of which is based on n-grams. They first evaluated one of this detector with 5-Fold cross validation on 2 229 goodware and 128 malware and the other detector with 3 856 malware and 169 goodware. Finally, their complete approach called “McBoost” was evaluated with 5-Fold on 3 830 malware and 503 goodware.

Tahan, Rokach, and Shahar (2012) recently presented “Mal-ID”, a malware detector that relies on high-level features obtained with Static Analysis. Their experiments are performed with 10-Fold on a dataset built with 2 627 benign executables and 849 known malware.

[3] While 10-Fold is equivalent to testing 10 times on 10% while being trained on 90% of the dataset, 5-Fold is equivalent to testing 5 times on 20% while being trained on 80% of the dataset.

Approach	Year	Sources	Historical Coherence
DREBIN: Arp, Spreitzenbarth, Hübner, Gascon, and Rieck, 2014	2014	"Genome, Google Play, Chinese and russian markets, VirusTotal	No
Canfora, Mercaldo, and Visaggio, 2013	2013	"common Android Markets" for goodware, "public databases of antivirus companies" for malware	No
Sahs and Khan, 2012	2012	Undisclosed	No
DROIDMAT: Wu, Mao, Wei, Lee, and Wu, 2012	2012	Contagio mobile for malware, Google Play for goodware	No
Amos, Turner, and White, 2013	2013	Genome, VirusTotal, Google Play	No
Demme et al., 2013	2013	Contagio mobile and Genome for malware, Undisclosed for goodware	No
Yerima, Sezer, McWilliams, and Muttik, 2013	2013	"from official and third party Android markets" for Goodware, Genome for malware	No
Gascon, Yamaguchi, Arp, and Rieck, 2013	2013	Google Play (labels from 10 commercial antivirus scanners)	No

Table 3.1: A selection of Android malware detection approaches

3.3 Empirical studies & Performance Assessment

In this section, we propose to revisit related work to highlight the importance of our contributions related to methodology that will be described in depth in chapter 7. We briefly present previous empirical studies and their significance for the malware detection field. Then we go over the literature of malware detection to discuss the assessment protocols.

3.3.1 Empirical studies

Empirical studies have seen a growing interest over the years in the field of computer science. The weight of empirical findings indeed help ensure that research directions and results are in line with practices. This is especially important when assessing the performance of a research approach. A large body of the literature has resorted to extensive empirical studies for devising a reliable experimental protocol (Bissyandé et al., 2013; Jones & Harrold, 2005; Hutchins, Foster, Goradia, & Ostrand, 1994). Guidelines for conducting sound Malware Detection experiments were proposed by Rossow et al. (2012). Our work—in particular chapter 6, chapter 7 and chapter 8—follows the same objectives, aiming to highlight the importance of building a reliable assessment protocol for research approaches, in order to make them more useful for real-world problems.

In the field of computer security, empirical studies present distinct challenges including the scarcity of data about cybercrimes. On this matter, we refer the reader to a report by Böhme and Moore (2012). Recently, Visaggio *et al.* empirically assessed different methods used in the literature for detecting obfuscated code (Visaggio, Pagin, & Canfora, 2013). Our work is in the

same spirit as theirs, since we also compare different methods of selecting training datasets and draw insights for the research community.

With regards to state-of-the-art literature tackled in this work, a significant number of Machine Learning approaches for malware detection (Gascon, Yamaguchi, Arp, & Rieck, 2013; Arp, Spreitzenbarth, Hübner, Gascon, & Rieck, 2014; Aafer, Du, & Yin, 2013; Barrera, Kayacik, van Oorschot, & Somayaji, 2010; Chakradeo, Reaves, Traynor, & Enck, 2013; Peng et al., 2012) have been presented to the research community. The feature set that we use in chapter 7 is thoroughly evaluated in chapter 6 and achieved better performance than those approaches. Thus, our experiments are based on a sound feature set for malware detection. We further note that in the assessment protocol of all these state-of-the-art approaches, the history aspect was eluded when selecting training sets.

3.3.2 Malware Detection & Assessments

We now review the assessment of malware detection techniques that are based on machine learning. For comparing performances with our own approach described in chapter 7, we focus only on techniques that have been applied to the Android ecosystem. In Table 3.1, we list recent "successful" approaches from the literature of malware detection, and describe the origin of the dataset used for the assessment of each approach. For many of them, the applications are borrowed from known collections of malware samples or from markets such as Google Play. They also often use scanners from VirusTotal to construct the ground truth. In our approach, we have obtained our datasets in the same ways. Unfortunately, to the best of our knowledge and according to their protocol descriptions from the literature, none of the authors has considered clearly ordering the data to take into account the history aspect. It is therefore unfortunate that the high performances recorded by these approaches may never affect the fight against malware in markets.

A recent achievement of machine learning-based malware detection for Android is DREBIN (Arp et al., 2014). The authors of this approach have relied on different features from the manifest file as well as the bytecode to build its SVM classifiers. The performance recorded in their paper are comparable to the performances obtained with our classifiers built during history-unaware 10-Fold experiments. DREBIN goes further by providing explainable outputs, i.e., being able to somehow justify why an app was classified as malware. Unfortunately, these performance records might have the effect of a sword cutting through water since the authors do not mention taking into account real-world constraints such as the relevance of history.

None of the approaches presented here has indeed taken into account the history aspect in their assessment protocol.

Part II

Exploring Android Applications

Chapter 4

Collecting a Dataset

Contents

A requirement to perform malware detection experiments is to have access to a collection of applications, both malware and goodware. While a small collection of known malware was released in 2012 (Zhou & Jiang, 2012), no collection of known goodware was ever published.

We decided to collect a large collection of Android applications and hence set up an infrastructure to automatically gather applications, and to perform various analyses on them.

4.1 Crawling application repositories

We have developed specialised crawlers for several market places to automatically browse their content, find Android applications that could be retrieved for free, and download them into our repository.

We have found that several market owners took various steps in order to prevent their market from being automatically mined. Thus, for such markets, we cannot guarantee that we have retrieved their whole content. However, to the best of our knowledge, the total number of apps that we have collected constitutes the largest dataset of Android apps ever used in published Android research studies.

Often, it is impossible to know beforehand how many apps are available on a given market. Therefore, some of the markets for which we wrote dedicated crawlers proved to be much smaller than initially expected.

The crawlers we wrote follow two main objectives: a) Collect as many apps as possible, and b) Ensure the lowest possible impact on the market infrastructure. These two objectives increased the cost of writing such crawlers since for every market a manual analysis of the website have been performed in order to detect and filter out pages with different URL but with similar contents—for example lists that can be sorted according to different criteria. Similarly, a unique identifier for every APK on one market had to be found, so that APK deduplication can happen *before* downloading the application.

While reducing the load we incur to markets' web servers may not seem strictly necessary to the objective of collecting apps, it vastly reduces the likelihood of being banned by markets' owners and hence, helps building and maintaining in the long run a large dataset.

4.2 Applications Sources

Google Play The official market of Android[1] is a web-site that allows users to browse its content through a web browser. Applications cannot however be downloaded through a web browser. Instead, Google provides an Android application[2] that uses a proprietary protocol to communicate with Google Play servers. Furthermore, no application can be downloaded from Google Play without a valid Google account – not even free Apps. Both issues thus outlined were overcome using open-source implementations of the proprietary protocol and by creating free Google accounts. The remaining constraint was *time*, as Google also enforces a strict account-level rate-limit. Indeed, one given account is not allowed to download more than a given quantity of applications in a given time frame.

Anzhi The anzhi market[3]—the largest alternative market of our dataset—is operated from China and targets the Chinese Android user base. It stores and distributes apps that are written in the Chinese languages, and provides a less-strict screening policy than e.g., Google Play.

AppChina AppChina[4], another Chinese market, used to enforce drastic scraping protections such as a 1Mb/s bandwidth limitation and a several-hour ban if using simultaneously more than one connection to the service.

4.2.1 Other Android Markets

The **1mobile**[5] market proposes free Android apps for direct downloads. It is a large market that offers users with opportunities of browsing and retrieving thousands of apps.

Other crawled markets are **AnGeeks** [6], and **Slideme**[7] wich is operated from the United States of America, and is a direct competitor of Google Play: Slideme provides both free and paid Apps for the Android platform.

FreewareLovers[8] is run by a German company, and provides freeware for every major mobile

[1] http://play.google.com (previously known as Google Market)
[2] Also named Google Play
[3] http://www.anzhi.com
[4] http://www.appchina.com
[5] http://market.1mobile.com
[6] http://www.angeeks.com
[7] http://slideme.org
[8] http://www.freewarelovers.com

platform, including Android. An advantage of FreewareLovers is that it does not require any specific application and can be used with any web browser.

ProAndroid[9], operated from Russia, is amongst the smallest markets that we crawled. It distributes free Apps only.

We also crawled **HiApk**[10] and **F-Droid**[11], a repository of Free and open-source software on the Android platform that provides a number of apps that users can download and install on their devices. Many of the applications found on F-Droid are modified versions of apps that are released to other markets by their developers. The modifications brought by F-Droid are usually linked with advertisement and/or tracking library removal.

4.2.2 Other sources

In addition to market places, we also looked into other distribution channels to collect applications that are shared by bundles.

Torrents We have collected a small set of apps which were made available through BitTorrent. We note that such applications are usually distributed without their authors' consent, and often include Apps that users should normally pay for. Nevertheless, when considering the number of leeches, we were able to notice that such collections of Android applications appear to attract a significant number of user downloads, increasing the interest for investigating malware distributed in such channels.

Genome[12] Zhou et al.Zhou and Jiang, 2012 have collected Android malware samples and gave the research community access to the dataset they compiled. This dataset is divided in families, each containing malware that are closely related to each other.

4.3 Architecture of the Crawling and analysis platform

Several software components were developed to build our collect and analysis infrastructure.

[9] http://proandroid.net
[10] http://www.hiapk.com
[11] http://f-droid.org
[12] http://www.malgenomeproject.org

4.3.1 Crawlers

For most of application sources, we developed a dedicated web crawler using the scrapy[13] framework. Every candidate application which is free runs through a processing pipeline that:

1. Ensures this app has not already been downloaded;
2. Downloads the file;
3. Computes its SHA256 checksum;
4. Saves the file.

To check that the application has not been downloaded already, each crawler obtains an APK identifier local to the market it deals with, and stores in a CouchDB [14] base an entry market_name–App_identifier. As a consequence, and because it is impossible to determine that two files from two markets are the same unless both are downloaded and compared, the deduplication is local to one market, meaning that one file from one market is downloaded exactly once, regardless of whether or not is has already been downloaded in another market.

4.3.2 Google Play Crawler

The official Google Play market, because of its specific characteristics, was handled in a different, more elaborate way. Indeed, Google Play has several features that make automatically crawling it harder than other markets. Amongst those features are the need for authentication with a valid Google Account currently associated with an Android device, the impossibility to obtain a list of all available applications and the necessity to use an undocumented protocol for communicating with Google Play servers. Furthermore, Google enforces limits on the number of apps that can be downloaded in a given period from one IP address, but also from one Google account.

To overcome those limits, we wrote a software dedicated to finding and downloading apps from Google Play. This software is built with two components: a central dispatcher, and a download agent.

[13] http://scrapy.org
[14] https://couchdb.apache.org

4.3.2.1 Central Dispatcher

The first part of this software ensures the coordination between the download agents. It provides agents with a Google account, and allows agents to request for another account if the first account encounters too many errors, most probably because it became blacklisted.

The dispatcher also instructs agents to either search for new APKs or to download APKs that were found as a results of a previous search.

Another role of the dispatcher is to maintain the list of new APKs that were found, and to ensure their deduplication so that one new application discovered by two different agents only is downloaded once.

Finally the dispatcher gives agents a list of ten applications to download and updates the list of newly discovered applications according to the success or failure of their download.

The dispatcher is written in the Python language, using the Flask micro web framework[15]. It relies on a PostgreSQL[16] relational database for its data storage needs, and communicates with agents using JSON messages by receiving and replying to HTTP requests.

4.3.2.2 Download Agent

Designed to have very few dependencies in order to ease its deployment, the Download Agent leverages googleplay-api[17], a Free implementation of the protocol used to communicate with Google Play.

The agent is also in charge of picking random words from stemmed English and French dictionaries, and then of performing search requests based on those words. This is a similar technique to that used in the Playdrone study published in 2014 by Viennot, Garcia, and Nieh (2014).

Searching applications is a useful step because when browsing applications by Categories, Google Play only presents the 500 most popular apps of this category, preventing us to obtain the long-tail of free applications that did not manage to attract a large audience.

After every request, agents wait a number of seconds to try to avoid being detected by Google anti-crawling measures.

[15] http://flask.pocoo.org
[16] http://www.postgresql.org
[17] https://github.com/egirault/googleplay-api

4.3.2.3 Deployment & Performance of our Google Play crawler

We have used agents on up to seven machines located in Luxembourg, France and Canada. On three of these machines, we ran two instances of the agent, one using exclusively IPv4 connectivity and the other using IPv6. Because IPv4 and IPv6 addresses are not linked in any way, this allows to hide the fact that those two agents run on the same machine, hence enabling us to increase the number of applications downloaded from one computer without increasing the risk of being blacklisted.

Our Google Play Crawler infrastructure managed to collect up to 296 448 new APKs in just one civil week, which demonstrates the ability of our software easily cope with the volume of free applications published through the official market.

After several weeks catching up with old applications, it appeared that two agents are sufficient to keep up with the flow of newly released apps.

4.3.3 Collection Manager

The collection manager is a web service responsible for all bookkeeping activities. It receives all the APKs that were downloaded by crawlers, and stores them on the file system, handling safely the potential conflicts inherent to every parallel software. It also keeps records of which market(s) each collected application has been seen in.

It enables applications to be downloaded by authenticated users, and also provides a web page presenting statistics on the whole dataset and on the recently added APKs.

This software component is written in Python using the web.py[18] framework. A PostgreSQL database accommodates data storage and querying needs, and embeds parts of the application's logic in PL/pgSQL functions.

4.3.4 Analysis Manager

To automatically perform per-application analyses on the collection, we created a program that monitors the collection, creates analysis tasks and saves the results of the analysis tasks.

[18] http://webpy.org

Our analysis manager is plugin-based so that we could easily add new analyses. It saves information about analysis failures, and about whether an analysis has already been performed on a given application. This helps managing the backlog induced by the addition of a new analysis plugin when the collection already contains applications.

The results of analyses are written to either a PostgreSQL database or to a CouchDB NoSQL database.

The tasks generated are sent to analysis workers, and the results are received from workers through several queues managed by a RabbitMQ[19] messaging server, enabling easy distribution of the tasks amongst many workers.

4.3.5 Analysis Worker

While the analyses manager determines what tasks need to be performed, the workers are in charge of actually performing the analyses.

Given the number of applications in the collection, and the number of analyses we envisioned, we had to be able to distribute the workload to many computers. With our RabbitMQ-based task distribution system, we can easily adapt the number of workers to the current workload. In particular, when adding a new analysis plugin to an already large collection, backloging creates a huge spike on the task count. In such occasions, we simply launched temporary workers until the task queues came back close to being empty.

On average, having a dozen workers—each using one CPU core—was enough to keep up with task creation rate. We have at several occasions launched two hundreds workers, leveraging the University of Luxembourg High Performance Computing[20] resources, and have observed that the task done per unit of time increases linearly with the number of workers, thus demonstrating the scalability of our approach.

Amongst other analysis modules, we wrote plugins to:

- submit applications to VirusTotal and collect the antivirus detection reports;
- extract compilation date, application's package name and version number;
- obtain certificates and their X.509 data, making possible the study presented in chapter 5;
- extract various representations of the bytecode, notably the list of basic blocks that is used in chapter 6 and chapter 7 of this thesis;

[19] https://www.rabbitmq.com
[20] https://hpc.uni.lu/

- derive a variety of application signatures, that could be used to conduct experiments on Android application similarity;
- extract package, class and method's names, enabling the detection of shared libraries.
- *etc*.

4.4 Challenges

In the course of keeping our collecting infrastructure running, we identified several difficulties. Besides those issues listed below, we also noticed two instances where a market would be unreachable for a period of time longer than any expected maintenance-induced downtime. For a full month, the 1mobile market was unavailable and then came back to normal. The market apk_bang however completely disappeared just a few days after we started crawling it, never to come back online again.

4.4.1 HTML Stability

During the time we collected applications, our crawlers had to be adapted around twenty times. Indeed, very often one market made changes to the structure of the HTML pages it generates. Most of the times, those changes implied that the XPath expressions used to scrap useful information from web pages had to be fully rewritten, which requires a new manual analysis of the web pages.

4.4.2 Monitoring Crawlers

Detecting that an HTML stability issue is happening may not always be straightforward. For the smaller markets, it is not unusual to detect no new application during several days. This can have two possible explanations: Either no new application was offered by one given market —in which case our crawler is working as expected—or it could be that our crawler failed to detect and/or collect the new applications —which could mean an HTML stability issue happened.

4.4.3 Protocol Change

One market moved from a standard website where applications could be downloaded from a web browser to a model where applications could only be obtained through a dedicated, market-specific application. While we probably could have reverse-engineered the undocumented protocol used by that market application, we considered that it was not worth the effort and instead simply stopped collecting apps from this market.

4.4.4 Information Loss

Very few application sources allow users to download previous versions of one given application. Instead most markets only allow the latest version to be downloaded. Coupled to the fact that it is not unusual for applications to be updated several times a week, it is impossible to guarantee that all versions of one application have been added to our collection. Moreover, if one version could not be downloaded before it was replaced by a newer one, it will never be available again.

4.5 Summary of our dataset

After more than two years of collecting, our dataset contains more than two millions unique Android applications, adding up to more than 14 TB. The distribution of applications according to their source is shown in Table 4.1.

We want to stress that this dataset was built over time. As a consequence, the experiments described throughout this thesis could only be based on the portion of our dataset we had collected at the time of experiment. This explains why each experiment uses a different number of applications. Every experiment presented in this thesis is accompanied by a description of the dataset that was used and that is always a subset of the whole dataset in its final state.

Table 4.1: Final state of our Application Repository

Marketplace	**# of Android apps**	**Percentage**
Google Play	1 489 572	70.33%
Anzhi	367 534	17.35%
AppChina	178 648	8.44%
1mobile	57 506	2.72%
AnGeeks	55 481	2.62%
Slideme	31 681	1.50%
torrents	5 294	0.25%
freewarelovers	4 145	0.20%
proandroid	3 683	0.17%
HiApk	2 453	0.12%
fdroid	2 023	0.10%
genome	1 247	0.06%
apk_bang	363	0.02%
Total	**2 117 825 Unique apps**	

Chapter 5

The Landscape of Android Malware

Contents

This chapter is based on work published in Allix, K., Jérome, Q., Bissyandé, T. F., Klein, J., State, R., & Le Traon, Y. (2014). A forensic analysis of android malware: how is malware written and how it could be detected? In *Computer software and applications conference (compsac)*

5.1 Preliminaries

An investigation into the business of malware requires a significant dataset representing real-world applications. We have built our dataset by collecting applications from *markets*, i.e., the on-line stores where developers distribute their applications to end-users. Indeed, although Google –the main developer of the Android software stack– operates an official market named *Google Play*, the policy of Google makes it possible for Android users to download and install Apps from any other alternative market.

Alternative markets are often created to distribute specific selection of applications. For example, some of these markets may focus on a specific geographical area, e.g., Russia or China, providing users with Apps in their local languages. Other markets focus exclusively on free software, and at least one market is known to be dedicated to adult content. Users may also directly share Apps, either in close circles, or with application bundles released through BitTorrent. Such apps are often distributed by other users who have paid for them in non-free markets. Finally, we have included in our datasets, apps that have been collected by others to construct research repositories.

In the remainder of this section, we provide details on the different sources of our dataset, on the scanning process that were used to label each application as malware or benign, and on the artifacts that we have extracted from application packages to perform our study.

Table 5.1 summarizes the number of applications collected from each market used to build our dataset. The largest share of applications are from the official Android market, Google Play. Using the SHA256 hash function on applications, we noticed that several thousands applications are found in more than one market. Hence, the total number of unique apps in Table 5.1 is less than the sum of unique applications in each market.

5.1.1 Artifacts of study

To perform our study we have mined information from the application packages focusing on two artifact metadata in Android package files.

Table 5.1: Origin of the Android apps in our dataset

Marketplace	# of Android apps	Percentage
Google Play	325 214	54.73%
appchina	125 248	21.13%
anzhi	76 414	12.86%
1mobile	57 506	9.68%
slideme	27 274	4.68%
torrents	5 294	0.89%
freewarelovers	4 145	0.70%
proandroid	3 683	0.62%
fdroid	2 023	0.34%
genome	1 247	0.21%
apk_bang	363	0.06%
Total	**594 000 Unique apps**	

Packaging dates An Android application is distributed as an `.apk` file which is actually a ZIP archive containing all the resources an application needs to run, such as the application binary code and images. An interesting side-effect of this package format is that all the files that makes an application go from the developer's computer to end-users' devices without any modification. In particular, all metadata of the files contained in the `.apk` package, such as the last modification date, are preserved.

All bytecode, representing the application binary code, is assembled into a `classes.dex` file that is produced at packaging-time. Thus the last modification date of this file represents the packaging time. In the remainder of this chapter, *packaging date* will refer to this date.

Certificate Metadata In the Android platform, a first security measure was made mandatory to guarantee that the authenticity of each application can be traced back to its creator. Thus, all Android applications must be signed with a cryptographic certificate. Certificates are included in the app package to allow end-users to verify the package's signature. For each application from our dataset, we have collected the certificates and analyzed their attributes, including *owner* and *issuer*, as described by the X.509 standard ITU, 2008.

5.1.2 Malware Labelling

Over the course of several months, while we collect the dataset, we have undertaken to analyze them with antivirus products actually used in the software market. For our study, we have relied

on VirusTotal[1], a web portal that hosts about 40 products from renown antivirus vendors, including McAfee®, Symantec® or Avast®. We have sent all applications from our dataset to VirusTotal and collected the scan results for analysis and correlation studies.

5.1.3 Test of Statistical Significance

Our forensics analysis is based on a sample of Android applications. Although, to the best of our knowledge, no related study involving Android malware has ever exploited that many applications, there is a need to ensure, for some of our findings, that they are significant. To this end, we resort to the common metric of the Mann-Whitney-Wilcoxon (MWW) test.

The MWW test is a non-parametric statistical hypothesis test that assesses the statistical significance of the difference between the distributions in two datasets Mann and Whitney, 1947. We adopt this test as it does not assume any specific distribution, a suitable property for our experimental setting. Once the Mann-Whitney U value is computed it is used to determine the p-value. Given a significance level $\alpha = 0.001$, if $p - value < \alpha$, then the test rejects the null hypothesis, implying that the two datasets have different distributions at the significance level of $\alpha = 0.001$: there is one chance in a thousand that this is due to a coincidence.

5.2 Analysis

In this section, we describe and interpret the results of our findings on how malware are written, in comparison with benign applications, and how antivirus products perform in their detection.

5.2.1 Malware identification by antivirus products

Malware identification by antivirus products is critical to practitioners and researchers alike. Indeed, antivirus products remain the most trusted means to flag an application as malware. Traditionally, the common detection scheme of antivirus is signature-based. Thus, to identify malware statically, antivirus software compares the contents of application files to their secret dictionary of virus signatures. This approach can be very effective, but can only help identify malware for which samples have already been obtained and associated signatures created. Some

[1] http://www.virustotal.com

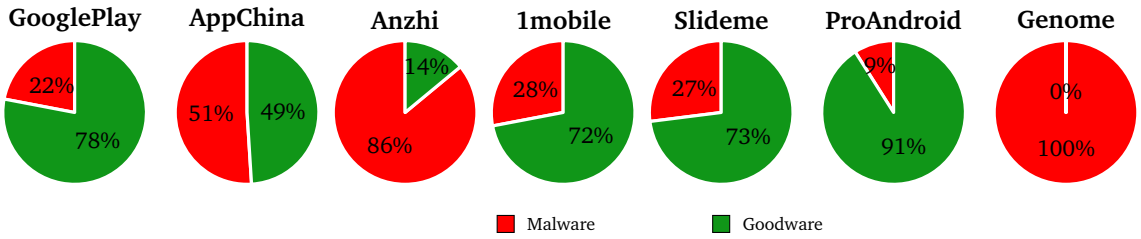


Figure 5.1: Share of Malware in Datasets: Applications are flagged by at least 1 antivirus product

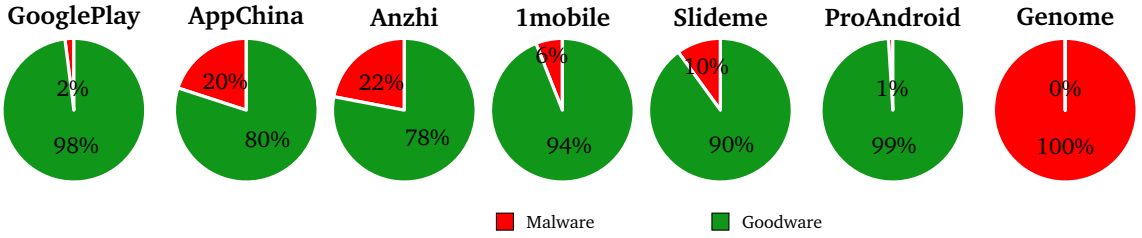


Figure 5.2: Share of Malware in Datasets: Applications are flagged by at least 10 antivirus products

antivirus products add heuristics to their process in order to identify new malware or variants of known malware.

In Figure 5.1, we see that most of our data sources contain Android applications that are flagged as malware by at least 1 antivirus product hosted by VirusTotal. Even Google Play, where each application goes through the Bouncer[2], shows a malware-rate of 22%. These malware are often in the form of adware, i.e., applications that continuously display undesired advertisement during use. Anzhi and AppChina include the largest share of flagged applications. Each of all

[2]Google's in-house environment for screening malware

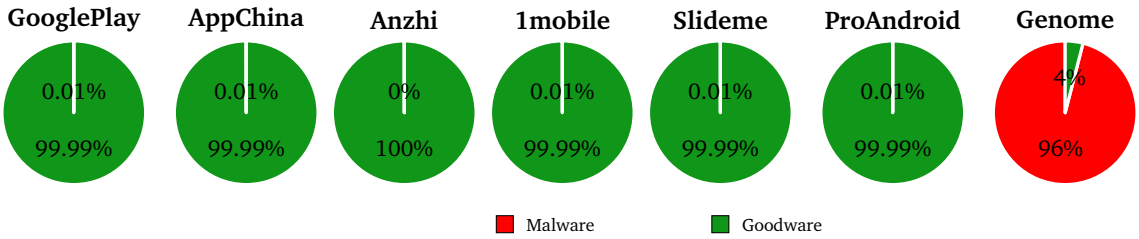


Figure 5.3: Share of Malware in Datasets: Applications are flagged by at least 26 antivirus products

the malware samples from the Genome dataset are indeed flagged by at least one antivirus software.

Malware shares depicted in Figure 5.2 indicate that antivirus software have divergent scanning results. Indeed, if we require that an application should be tagged as a malware only if at least 10 antivirus products have found it suspicious, then the malware rate drops significantly for all our data sources. Google Play now only contains 2% of malware, while all Genome samples are still identified as true malware.

The Genome dataset being a reliable source of known malware, we change the threshold of antivirus until some of the applications in the dataset are missed in the scanning process. Figure 5.3 provides the different malware share when at least 26 antivirus, out of more than 40, are required to flag an application before it is considered a malware.

> Antivirus software cannot each identify all existing malware. Only a small subset of widely known malware are recognised by a large number of antivirus software.

5.2.2 Android Malware Production

We proceed to investigate the production of Android malware to draw insights. The analysis of packaging dates of Android applications yields some distinct patterns. In Figure 5.4, we plot the packaging date, subdivided by hour, for benign applications and for all applications flagged by at least one antivirus. Despite the potential noise due to the threshold set by each antivirus to tag malware, we note a pattern in the compilation dates: it stands out that there are many more peaks of malware packaging. This suggests that malware often are compiled in batches, while compilation of benign applications are more spread over time.

To further investigate and strengthen the validity of our finding, we consider the samples of confirmed malware from known families exposed in the Genome dataset, and consider all other applications from our datasets as benign. This process is valid when considering a very strict threshold where an application is labelled as malware if at least half, i.e., 22, of the antivirus software from VirusTotal flag it. Figure 5.5 thus confirms more strongly that Android malware are compiled in batches. The 1258 malware of the genome dataset have been packaged on only 244 different days. 51 malware were packaged on 2011-09-21 alone, representing 16% of all Android apps packaged on this day. Only 72 malware were packaged each alone in a distinct day when no other malware was packaged. We counted 78 cases where at least two malware were packaged in the same second. At 15 instances, four or more malware were packaged in the same second; Two of those instances saw ten or more new malware being packaged. Such a strong time locality suggests that malware writers have set up tools to automate the malware packaging

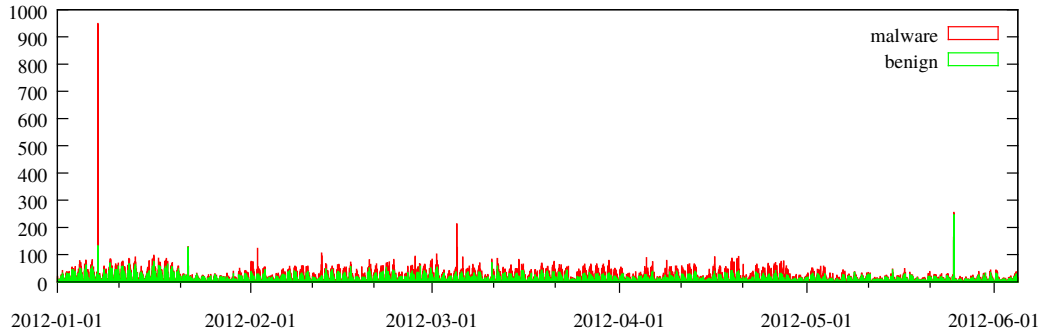


Figure 5.4: Number of benign and malware packaged between 01 January 2012 and 01 June 2012

process. One single certificate (md5: *264BF7D71E0EDC4FCB8A9A16AB7C3357*) even managed to sign 781 apps detected as malware by at least one antivirus in the same second (2012-01-07 14:25:06).

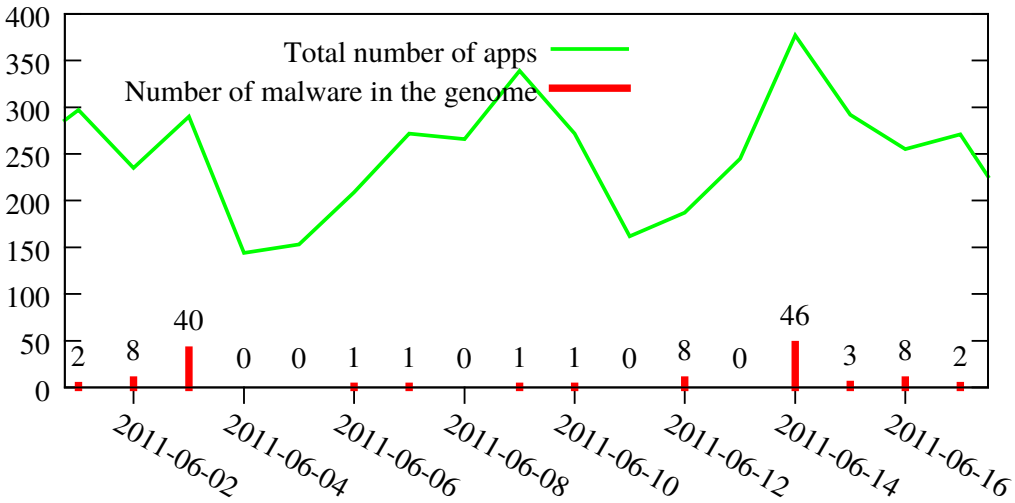


Figure 5.5: Number of packaged application and of packaged malware over time: Focus on period 2011-06-01 to 2011-06-17

Malware development is often a standardized process that aims at producing a large number of malware at once. Aside from rare cases of target-specialized malware, malware are built in bulk in the like of slightly different applications.

Table 5.2: Distribution of Android packaging dates across week days

	Monday		Tuesday		Wednesday		Thursday		Friday		Saturday		Sunday	
Benign Apps	56,476	15.75%	57,728	16.10%	58,078	16.20%	58,995	16.45%	58,926	16.43%	34,223	9.54%	34,182	9.53%
Malware *(Threshold=24)*	236	14.68%	376	23.38%	284	17.66%	276	17.16%	225	13.99%	90	5.60%	121	7.52%
Malware *(Threshold=25)*	211	14.46%	342	23.44%	265	18.16%	254	17.41%	205	14.05%	81	5.55%	101	6.92%
Malware *(Threshold=26)*	200	14.60%	328	23.94%	249	18.18%	234	17.08%	190	13.87%	74	5.40%	95	6.93%

5.2.3 Business of Malware Writing

We now look into the process of malware writing. We focus our analysis on their apparition cycles by clustering Android applications based on the week day during which they were packaged. For this experiment, we only consider malware that were detected by at least half of the antivirus software operated by VirusTotal.

Table 5.2 highlights the distribution of app packaging dates for both benign applications and malware across week days. The percentage of apps packaged during business days are actually similar for malware and benign applications. A test of significance with the MWW test further confirms that the statistical difference is near null.

On average, 19% of benign applications are packaged during weekends, while this is the case for only 13% of malware. We further use the MWW test to confirm that the difference between weekdays and week-end days is statistically more significant for malware than for benign. There is thus a clear pattern of five-day work per week. A possible explanation to this pattern could be that malware writing is performed by some developers during their regular office hours while working for their employer. A second reason might be that malware writers follow a standard work schedule and do not work during weekends, thus suggesting an industrial process in the building of malware rather than a spare-time hobby.

> There appear to be evidence that the business of malware writing, or at least their proliferation, is at an industrial scale.

5.2.4 Digital Certificates

Android applications rely on digital certificates to build a trust model between developers and end users. Applications signed using the same certificate can share information and data at runtime (if allowed by explicit permissions). Certificates also allow to link a set of applications with their developer, although this linking does not ensure that the identity of a developer is certified.

Indeed, certificates can be self-signed, rather than signed by a competent trustworthy authority, and therefore do not necessary lead to the real developer. However, finding the same certificate (serial number, fingerprint , issuer and owner) in several applications is a strong indicator of either a unique origin, or of advanced certificate stealing and reuse.

Our analysis on certificates aims at understanding the practice of certificate use by malware writers. We first note, based on our datasets, that self-signed certificates are the norm rather than the exception for Android developers. Of the 165 542 certificates in our dataset, only 51 are not self-signed. Self-signed certificates were used to sign 99.88% of the apps in our dataset. Consequently, most certificates carry no information that could be trusted about the identity of the application developer.

Our findings apply particularly to malware development. We focus our study on the subset of malware in the Genome dataset. These are well established malware that most antivirus products can identify. For instance, the certificate that holds the serial number `E6EFD52A17E0DCE7` was used in at least two different malware applications. Manual searching for the Issuer-related fields[3] does lead to the blog of a well known Android developer. One entry of this blog addressed the issue of signing of Android applications. After reading this entry, we found out that writers of the referred malware just copy/pasted the command in the posted example without any effort to change the basic information that indicates what a certificate is supposed to certify.

We have further investigated this copy/paste strategy and found that it occurs too often. Thus, although a certificate issued to Android Debug can be used to develop and test an application, the release version cannot be published with such a certificate. This basic rule is stated in almost every online tutorial and Android textbook. Yet, we identified more than 50 well-known malware which use such a certificate: this questions the competency or may highlight the laziness of malware writers as in a day-to-day job.

Finally, our manual investigation into the attributes of certificates in malware, reveal that, sometimes, malware writers brag or use obvious offensive names. For instance, a certificate whose *owner* is named *PhoneSniper* appears in at least 281 different malware. If users were able to carefully inspect certificates before installation, such malware would have been less propagated. Similarly, this information could be used with techniques of natural language processing to silently filter some malware in application markets.

The vast majority of Android apps in our datasets are signed with a certificate that was used to sign very few other applications. Indeed, we have found that 95% of the certificates signed less than 10 apps. However, for the remainder of certificates that were used in large numbers of applications, different patterns emerge.

[3] `Issuer: C=ID, ST=Jawa Barat, L=Bandung, O=Londatiga, OU=AndroidDev, CN=Lorensius W. L. T/emailAddress=lorenz@londatiga.net`

Table 5.3: Top 20 certificates which were used to sign the most malware

Certificate MD5	Number of Benign	Number of Malware	Certificate Issuer & Owner
E8...87	4 623	192	C=US, ST=California, L=Mountain View, O=Android, OU=Android, CN=Android/emailAddress=android@android.com
E5...3F	0	167	C=keji0003
CF...26	1	166	C=cn, ST=shenzhen, L=china, O=Phone, OU=Phone, CN=PhoneSniper
50...BA	0	98	C=kejikeji, ST=kejikeji, L=kejikeji, O=kejikeji, OU=kejikeji, CN=kejikeji
E5...C2	0	95	C=US, OU=Google Inc.
8B...D2	0	52	CN=Fujian Kaimo Network Tech
3C...3E	0	29	C=a, ST=a, L=a, O=a, OU=a, CN=a
AC...A7	1	21	CN=Sexy
C4...2B	0	20	C=CA, ST=Ontario, L=Toronto, O=Typ3 Studios, OU=Typ3 Studios, CN=Typ3 Studios
CF...6C	0	19	C=0
1D...07	6	17	C=CN, ST=Sichuan, L=Chengdu, O=jiemai-tech, OU=jiemai-tech, CN=Jiemai Technology
77...F3	8	17	CN=alan
B1...A4	0	17	OU=Safe System Inc., CN=Safe System Inc.
74...50	0	16	C=cn, ST=guangdong, L=shenzhen, O=hynoo, OU=hynoo, CN=wang
21...37	2	15	C=cn, ST=fujian, L=xiamen, O=guopai, OU=guopai, CN=jtwang
76...A8	1	14	C=CN, CN=picshow1
AC...94	0	13	C=86, ST=BeiJing, L=BeiJing, O=Gold Dream Studio, OU=Gold Dream Studio, CN=Hong Fu
73...A3	0	12	C=001, ST=US, L=LSA, O=www.android.com, OU=www.android.com, CN=Android
C6...1B	0	12	C=86, ST=SH, L=CN, O=MJ, OU=MJ, CN=MJ
E7...AE	34	12	C=0086, ST=Beijing, L=Beijing, O=Gall me, OU=Android, CN=Gall me

Table 5.3 summarizes the top certificates used in malware packages. Once again, we consider as malware all applications that were flagged as suspicious by at least half of the antivirus products in VirusTotal. The numbers distinctly provide evidence that there a mass development and deployment of Android benign and malware apps was put in place. For instance, three certificates were used each for more than 160 malware. The top-used certificate by malware is also used by over 4 623 benign applications: a realistic hypothesis to support this fact would be that the private key was somehow leaked, leading to many otherwise unrelated writers to use and share the same certificate.

We further consider the overlap between benign and malicious applications that share the same certificate. In Table 5.4, we indicate the top certificates that are used by both malware and goodware. We note that there is a clear overlap showing the usage of certificates for both malicious and benign applications. A number of explanations can be provided for this phenomenon:

- *Dr Jekyll and Mr Hyde syndrome*. Developers use the same development tools and environment for both legitimate and malicious applications. This observation supports the 5 working day behavior shown in table 5.2. This means that developers write malware during their regular working hours.

- *Reputation biasing*. In this hypothesis, a developer might increase her/his reputation by developing benign applications. As soon as enough positive reviews have been obtained, successive malware might be more easily downloaded and installed. For instance, the certificate with the serial 4DFF5300, has been observed signing both a malicious and a non malicious application on 2011-08-30, in the very same time: 21:52:38. On the overall 1 benign application and 176 malware are associated with this certificate. The most recent application in our dataset using this certificate was packaged on 2012-03-11 17:19:54, while its first usage can be traced back to 2011-07-14 21:45:12.

- *Antivirus false negatives*. Probably, some of the applications tagged as benign are in fact malicious. It is possible that existing tools have not detected them yet as malicious, due to a better obfuscation and stealthier behavior.

- *Antivirus false positives*. Antivirus can also wrongly flag a benign application as malware. For instance the digital certificate whose md5 is *75BDB3531C04EB8246846532A7AE2050* has been observed to sign 2 844 total applications, only one (1) of which being tagged as malicious. In this case, we suspect that either the certificate was stolen, but using it for only one single malicious application does not really make sense. More probable is the hypothesis that the single malicious application is a false positive. We have correlated this information also with the time-line of the packaging dates for this certificate. The single malicious application was packaged on 2013-11-15 19:16:04; On this very same day, this certificate signed 55 other apps that are all undetected by antivirus products. The

Table 5.4: Top 15 certificates which were used to sign many malware and which signed benign apps as well

Certificate MD5	Number of Benign	Number of Malware	Certificate Issuer & Owner
E8...87	4 623	192	C=US, ST=California, L=Mountain View, O=Android, OU=Android, CN=Android/emailAddress=android@android.com
1D...07	6	17	C=CN, ST=Sichuan, L=Chengdu, O=jiemai-tech, OU=jiemai-tech, CN=Jiemai Technology
77...F3	8	17	CN=alan
21...37	2	15	C=cn, ST=fujian, L=xiamen, O=guopai, OU=guopai, CN=jtwang
E7...AE	34	12	C=0086, ST=Beijing, L=Beijing, O=Gall me, OU=Android, CN=Gall me
8D...F9	92	10	C=US, ST=California, L=Mountain View, O=Android, OU=Android, CN=Android/emailAddress=android@android.com
DE...92	3	9	C=CN, ST=Guangdong, L=Guangzhou, O=synkay, OU=sunkay, CN=sunkay
C7...80	56	8	C=US, ST=Fl, L=Miami, O=Gp Imports, OU=Gp Imports, CN=Gp Imports
69...A5	87	7	C=CN, ST=beijing, L=beijing, O=Wali, OU=Wali, CN=Lee
34...F5	2	6	C=KR, ST=South Korea, L=Suwon City, O=Samsung Corporation, OU=DMC, CN=Samsung Cert/emailAddress=android.os@samsung.com
3D...10	6	4	CN=Ngan Viet Dung
BA...26	48	3	C=CN, ST=Zhejiang, L=Hangzhou, O=Feelingtouch, OU=Feelingtouch, CN=Feelingtouch
82...C5	2	3	C=86, ST=china, L=ysler, O=ysler, OU=ysler, CN=ysler.com
59...EE	178	2	C=86, ST=Guangdong, L=Guangzhou, O=3g.cn, OU=GAU, CN=Jarod Yv
51...B3	7	2	C=CN, ST=ShenZhen, L=ShenZhen, O=nmting.com, OU=nmting.com, CN=Ale Zhao

usage pattern for this certificate shows very frequent application signing, often with just a few minutes between two apps, and the application detected as a malware exhibits no deviation from this pattern. Furthermore, it would make sense for the developer to create a new certificate if he once wrote a malware, in order to avoid having his/her future benign applications signed with a certificate that is associated with a malware.

Malware writers do not use digital certificates properly, and often reuse compromised keys that were used to build certificates of benign applications.

5.3 Discussion

The forensic analysis that we have performed and whose results were outlined in the previous section has yielded a number of insights for the research and practice of malware detection. In this section, we summarize these insights and discuss how this empirical study could be instrumented in our work on malware detection.

5.3.1 Summary of findings

On Antivirus software Our large-scale analysis of hundreds of thousands of Android applications with over 40 antivirus products have revealed that most malware are not simultaneously identified by several antivirus. Only a small subset of common malware is detected by most antivirus software. This finding actually supports the idea that there is a need to invest in alternative tools for malware detection such as machine-learning based approaches which are promising to flag more malware variants.

On malware business We have presented empirical evidence that malware were mass produced. This raises a number of questions leading to hypothesis on how malware developers manage to remain productive. The first hypothesis would be that, malware is not written from scratch, thus providing an opportunity to detect malware by discovering the piece of code that was grafted to existing, potentially popular, apps.

5.3.2 Insights

Building a naive antivirus software Exploring the rate of shared certificates within malware, we were able to devise a naive malware detection mechanism based on the appearance of a tagged certificate. In its simplest form, the scheme consists in tagging any application as malicious if the signing key has been already observed for a confirmed malicious application.

To assess this naive approach we have considered that in a first phase we have manually discovered all malware packaged before 01/Jan/2013 in our dataset. We consider for this step only malware that are detected by at least half of the antivirus products. Then based on the certificates recorded for the found malware, we arbitrarily tag as malicious all applications packaged after 01/Jan/2013 and that are signed with any of the flagged certificates. Table 5.5 provides the results for this experiment. We were able to build a malware detector with a Precision of 84% (2,166 false positives out of $2,166 + 11,460$ tagged). While we succeed in flagging almost 1 actual malware out of 10, we only wrongly tag as malicious about 1 benign app in 100.

Table 5.5: Performance of a naive antivirus software based on certificates

	Benign apps tagged	**Malware tagged**
Number	2 166	11 460
Percentage	1.19%	8.82%

At the minimum, the obtained results show that our naive approach could be used by antivirus vendors to improve their recall, by being suspicious of more apps, and improve precision by trusting apps signed with certificates that have been used in a large number of benign apps.

Localizing malware Our findings on the potential mass production of malware could be leveraged in an approach of malware localization. Indeed, simultaneous development and packaging of malware suggests a redundant insertion of malware code in all applications. Thus, a similarity measure of the bytecode could allow to isolate this code and then locate it in other malware samples.

5.4 Conclusion

The recent and steady rise of Android malware over the past four years has lead to a rapidly growing automation in the malware creation process. Due to the specific nature of development

of Android applications, important artifacts leak out and can provide some insights about their creators. We have analyzed the available data through this perspective. For our large-scale study, we have considered over 500,000 Android applications, which included both benign applications and malware.

Packaging dates show substantial time localization behavior. Waves of packaging can be observed thus shedding a new light on the malware creation process. Digital certificates, albeit self-signed also provide valuable pieces of information. We have observed huge quantities of malware sharing the same private key and thus proving that either keys have been stolen, or those malware have the same origin. On the other hand massive copy/paste coding, relying on directly copying code from popular tutorials and blogs, shows that the malware programming is done at a fast pace by developers lacking elementary cryptography knowledge.

This, unfortunately shows that current Android malware as well as mitigation techniques are still in the infancy. It's surprising to see that most malware writers do not use digital certificates properly and that many of the current mitigation techniques did not check them. However, more troubling is the extent to which private keys seem to have been compromised and that both benign applications and malware share the same certificates.

In the future, we plan to leverage the insights discussed in Section 5.3. Furthermore, we plan to extend this work by considering also the automated analysis of the bytecode. Some preliminary work have been done and the results are promising.

Part III

The Art of Evaluating Malware Detectors

Chapter 6

The Gap Between *In-the-Lab* and *In-the-Wild*

Contents

This chapter is based on work published in Allix, K., Bissyandé, T. F., Jerome, Q., Klein, J., State, R., & Le Traon, Y. (2014). Empirical assessment of machine learning-based malware detectors for android: measuring the gap between in-the-lab and in-the-wild validation scenarios. *Empirical Software Engineering*, 1–29. doi:10.1007/s10664-014-9352-6 and in Allix, K., Bissyandé, T. F., Jérome, Q., Klein, J., State, R., & Le Traon, Y. (2014). Large-scale machine learning-based malware detection: confronting the "10-fold cross validation"scheme with reality. In *Proceedings of the fifth acm conference on data and application security and privacy* (pp. 163–166). CODASPY '14. San Antonio, Texas, USA: ACM. doi:10.1145/2557547.2557587

6.1 Introduction

Machine learning techniques, by allowing to sift through large sets of applications to detect malicious applications based on measures of similarity of features, appear to be promising for large-scale malware detection (Henchiri & Japkowicz, 2006; Kolter & Maloof, 2006; Zhang et al., 2007; Sahs & Khan, 2012; Perdisci et al., 2008a). Unfortunately, measuring the quality of a malware detection scheme has always been a challenge, especially in the case of malware detectors whose authors claim that they work "in the wild". Furthermore, when the approach is based on machine learning, authors often perform a 10-Fold cross validation experiment on small datasets to assess the efficiency of the approach. This combination of 10-Fold Cross Validation and small dataset is what we call an *in the lab* scenario. However, we claim that, in the field of malware detection, all the underlying hypotheses associated with an *in the lab* experiment must be outlined to allow a correct interpretation of the results. Indeed, validation experiments of malware detection approaches are often controlled and the datasets used may not be representative, both in terms of *size* and in terms of *quality*, of the targeted universe.

The present chapter is both an illustration and a complement to the study published by Rossow et al. (2012) and called "Prudent Practices for Designing Malware Experiments: Status Quo and Outlook". Our work focuses on realistic empirical assessment, one of the many issues raised by Rossow et al. In their introduction, they state:

> [...] we find that published work frequently lacks sufficient consideration of experimental design and empirical assessment to enable translation from proposed methodologies to viable, practical solutions. In the worst case, papers can validate techniques with experimental results that suggest the authors have solved a given problem, but the solution will prove inadequate in real use.

Indeed, while most of the studies presented in our related work section (3.3) were published *after* the paper of Rossow et al., they all present this very shortcoming in their validation methodology.

This chapter. We discuss in this paper a new machine learning-based malware detection approach that is effective when assessed with the *in the lab* validation scenario. However, our work aims at shedding light on whether *a high performance recorded with a typical* in the lab *experiment guarantees even a good performance in realistic malware detection use-cases*. To this end, we proceed to compare the performance of machine learning classifiers when they are being validated *in the lab* and when they are used in the wild (i.e., the way they are intended to be used). Due to the scarcity of author data and the lack of sufficient implementation details to reproduce approaches from the state-of-the art literature, we base our investigation on our newly designed malware detection approach. We have devised several machine learning classifiers and built a set of features which are textual representations of basic blocks extracted from the Control-Flow Graph of applications' bytecode. We use a sizeable dataset of over 50 000 Android applications collected from sources that are used by authors of state-of-the art approaches.

The contributions of this chapter are:

- We propose a feature set for machine-learning classifiers for malware detection.
- We show that our implemented classifiers yield a high malware discriminating power when evaluated and compared with state of the art techniques from the literature. This *in the lab* evaluation is based on the 10-Fold cross validation scheme which is popular in the machine learning-based malware detection community.
- We demonstrate limitations of this validation scenario that is performed in the literature of malware detection. In particular, we show with abundant experimental data that 10-Fold validation on the usual sizes of datasets presented in the literature is not a reliable performance indicator for realistic malware detectors.

This paper is organised as follows. Section 6.2 discusses malware detection in the wild and highlights the associated challenges. We provide in Section 6.3 various information on the datasets of our experiments, the investigated research questions as well as the used evaluation metrics. Section 6.4 describes our approach of malware detection, exploring the variables that can be parameterized to tune the output of the machine learning process. Section 6.5 presents the assessment of our approach, highlighting its performance against state of the art approaches, but also showing its counter-performance in the wild. Section 6.7 discusses potential threats to validity. Related work is discussed in Section 3.3. Section 7.5 concludes and enumerates future work.

6.2 Malware Detection in the Wild

The market share of Android and its open source architecture has made it a primary target for malware attacks among mobile operating systems. In the official Android application store,

Google Play, up to 40 000 new applications are registered in a month according to AppBrain, 2013b. In this context, especially for alternative markets, it is important to devise malware detection approaches that are efficient in: (1) quickly identifying, with *high precision*, new malware among thousands of newly arrived applications, (2) classifying a large set of applications to expose its *entire* subset of suspicious ones.

Machine learning is a tool used in Artificial Intelligence to provide computers with capabilities for automatically improving themselves in the recognition of patterns. Machine-learning algorithms rely on selected features and training data to infer the commonalities that a group of searched items share and that discriminate them from the rest of the universe. The success of these algorithms therefore depend on the *relevance of the features* for discrimating between the group of searched items and the rest, and on *the quality of training data* for being unbiased and representative of the universe of items. In machine learning-based malware detection, there is a challenge to meet both requirements. Indeed, in the wild, i.e., in real-world scenarios, there are much more goodware than malware, and it is yet difficult to build a set of "perfect" goodware that does not contain a single malware. Consequently, validation of the performance of malware detectors should reflect these specificities. Indeed:

- Using small datasets of goodware and malware of similar size cannot guarantee a realistic assessment of a malware detector that is intended to be used in the wild.
- Blindly using a goodware set without properly validating that it does not contain malware will significantly bias the yielded results

6.3 Data Sources and Research Questions

In this section, we mainly present the datasets that are used to assess our malware detection approach as well as the different aspects that are evaluated.

6.3.1 Datasets

For our experiments we have used two sources of Android applications that are often used by researchers and practitioners of machine learning-based malware detection for Android. However, to the best of our knowledge our dataset is the largest ever presented in the Android malware detection literature. We make it available to the research community.

Building an Android market dataset. Google Play[1] is the main Android applications market available, and thus constitutes a unique source of relevant applications that are used and that reflects the state of Android application development. We have built a tool that automatically crawls and downloads free applications available in this source. Due to limitations in the implementation of our tool and to restrictions set by Google regarding automatic crawling, we could not retrieve all free applications. Nonetheless, in the course of six (6) months, we have collected a sizeable dataset of nearly 52 000 unique applications. Although Google use various tools to keep Google Play free of malware, we found, after investigation with antivirus, that our collected dataset includes malware.

Collecting known malware. For training needs, we must have access to a reliable and representative set of Android malware. To this end, we leverage a dataset released in the course of the Genome project by researchers from the North Carolina State University (Zhou & Jiang, 2012). The *Genome dataset* contains over 1 200 Android malware samples.

6.3.2 Research Questions

We now discuss four important research questions that we have formulated to assess the effectiveness of our machine learning-based malware detectors.

RQ1. *What is the sensitivity of the malware detector when the Goodware/Malware ratio changes in training data?* Because training data is an important element of a machine learning process, we investigate the impact of the composition of this data on the output of the malware detector.

RQ2. *How does the number of selected features influence the performance of the tool?* We study the correlation between the number of features used to discriminate malware and the performance of the malware detection scheme.

RQ3. *What is the impact of the underlying machine learning algorithm?* With this research question we want to assess that the algorithm that is used for the implementation of our approach does not significantly bias our findings.

[1] Google Play was formerly known as *Google Market*

RQ4. *What is the sensitivity of the tool towards the quality of training data?* In the wild, the supposed goodware dataset may be imperfect and contain unknown malware, hence adding noise to the training phase. We investigate the impact that such misrepresentations in training data can have to the final output of the malware detector.

Those four research questions contribute to the common goal of determining the performance of a malware detector for several sets of parameters. Indeed, evaluating a malware detector for one fixed set of parameters only tells the experimenter how it would perform under the exact same conditions.

6.3.3 Malware labeling.

For the purpose of guaranteeing a reliable assessment of our approach, we undertake to label all applications by classifying them beforehand as malware or goodware, thus building the ground truth. To construct a reference independent classification to which we can compare the predictions yielded by our machine learning-based approach, we collected from VirusTotal[2] the analysis report of each application in our datasets. VirusTotal is a service that allows security practitioners to readily obtain information on antivirus products which have identified a given application sample as malware. At the time of writing, VirusTotal supported around 40 different antivirus products which are continuously updated both in terms of software release version and in terms of malware databases. Several thousands of the malware in our datasets were unknown to VirusTotal before we submitted them.

6.4 Experimental Setup

Malware detection shares a few challenges with other field of computer science such as natural language processing where information retrieval techniques can be leveraged to isolate and retrieve information that is hard to see at first glance. For text classification (Jacob & Gokhale, 2007), researchers often rely on approaches based on n-grams, which, given a string of length M, are all the substrings of length n ($with\ n < M$) of this string. The difficulty in malware detection consists in recognizing, for classification purpose, the signature of a malware. Already in 1994, Kephart at IBM has proposed to use N-grams for malware analysis (Kephart, 1994). More

[2] https://www.virustotal.com

recently a large body of research in malware detection based on machine learning have opted for n-grams to generate file/program signatures for the training dataset of malware (Henchiri & Japkowicz, 2006; Kolter & Maloof, 2006; Santos, Penya, Devesa, & Bringas, 2009). Despite the high performance claimed by the authors for very small datasets, between 500 and 3 000 software programs, we believe that a malware detector based on n-grams, because of its vulnerability to obfuscation, could be trivially defeated by malware authors. For the Android platform, Sahs and Khan, 2012 recently proposed to use a combination of Android permission and a representation of programs' control-flow graphs. However, since all malware are not related to a permission issue, we believe that their approach will yield poor results for other various types of malware.

In this paper we propose a different approach to extract, from an application program, data blocks that are semantically more relevant for executed software. These blocks are elements of applications' Control Flow Graphs which should capture, in a more meaningful way than n-grams, the implementation of a malicious behavior inside program code.

6.4.1 Our Feature Set for Malware Detection

As detailed in previous sections, machine learning-based malware detection relies on a training data that is analyzed to *learn what could suggest that a given application is a potential malware*. To that end, the learning algorithm must be "told" what features are relevant in each piece of data of the dataset. Indeed, Machine Learning algorithms cannot work directly on Android applications; Each application must be represented with an ordered list of properties—called a Feature vector in the context of Machine Learning. Several sets of features designed to characterize executable code have been introduced in previous approaches (Cf. section 3.3).

Features are often extracted from program metadata or program code (binaries, bytecode, source code). In the case of the Android Operating System, features can be extracted from application bytecode using static analysis. Indeed, Android applications are distributed in the form of `.apk` files which are packages containing the application's *Dalvik*[3] bytecode, assets such as images, and metadata specific to the Android platform. Android applications are generally written in Java. The program is then compiled to Java bytecode which is converted into Dalvik bytecode. Unlike the typical binary code, Dalvik bytecode retains most of the information contained in Java bytecode. Thus, such code can be fed to Static Analysis tools that support Dalvik bytecode or after converting it back to Java Bytecode for which many analyzers exist. In our work, the static analysis was performed using AndroGuard.

We perform static analysis of Android applications' bytecode to extract a representation of the program control-flow graph (CFG). The extracted CFG is expressed as character strings using a

[3] Dalvik is a virtual machine that is included in the Android OS

method devised by Pouik *et al.* in their work on establishing similarity between Android applications (Pouik et al. 2012). This method is based on a grammar proposed by Cesare and Xiang (2010). This derived string representation of the CFG is an abstraction of the application's code that retains information about the *structure* of the code, but discards low-level details such as variable names or register numbers. In the context of malware detection, this is a desirable property. Indeed, two variants of a malware may share the same abstract CFG while having different bytecode. Thus, using an abstract representation of the code could allow to resist to basic forms of obfuscation, a threat to validity that n-grams-based approaches cannot readily overcome.

Given the abstract representation of an application's CFG, we collect all basic blocks that compose and refer to them as the features of the application. A basic block is a sequence of instructions in the CFG with only one entry point and one exit point. It thus represents the smallest piece of the program that is always executed altogether. By learning from the training dataset, it is possible to expose, if any, the basic blocks that appear statistically more in malware.

Let us note BB_i a basic block and BB_{all} the set of the n basic blocks encountered at least in one application.

$$BB_{all} = \{BB_1, BB_2, \cdots, BB_n\} \tag{6.1}$$

For every application App, we build a list, $Features_{App}$, of binary values $(0, 1)$ that codifies all basic blocks from BB_{all} that appear in the App and those that do not.

$$Features_{App} = (b_{App,1}, b_{App,2}, \cdots, b_{App,n}) \tag{6.2}$$

In Equation 6.2, $b_{App,i}$ is set to 1 if the basic block BB_i is present in the abstract CFG of App, and 0 otherwise.

Experimental analysis with all applications from our datasets have shown that with this method, we could extract over 2.5 millions different basic blocks, each appearing once or more in the CFGs of applications. The basic block representation used in our approach is a high-level abstraction of small parts of an Android application. Depending on its position inside a method, one sequence of instructions may lead to different bytecode because of register renumbering. Our abstract basic block representation however will always produce the same string for one sequence of instructions of a basic block, hence providing a higher resistance to code variations than low-level representations such as n-grams computed on bytecode. For reproducibility purposes, and to allow the research community to build on our experience, the feature matrices that we have computed for both the Genome and the Google Play dataset are publicly available for download[4].

[4] https://github.com/malwaredetector/malware-detect

6.4.2 Classification Model

Classification in machine learning-based approaches is the central phase during which an algorithm assigns items in a collection to target classes. In our case, the classification phase aims at predicting if a given application should be assigned to the malware class. In preparation to the classification phase, we must build a dataset in which the class assignments, i.e., goodware or malware, are known for the application. The classification model is then built by a classification algorithm which attempts to find relationships between the features of the applications and their class assignments. This process is known as the *training* phase of the algorithm. In our approach we rely on four (4) well-known classification algorithms, namely Support Vector Machine (SVM) (Cortes & Vapnik, 1995), the RandomForest ensemble decision-trees algorithm (Breiman, 2001), the RIPPER rule-learning algorithm (Cohen, 1995) and the tree-based *C4.5* algorithm (Quinlan, 1993).

We now discuss the different steps, illustrated in Figure 6.1, for building the classification model.

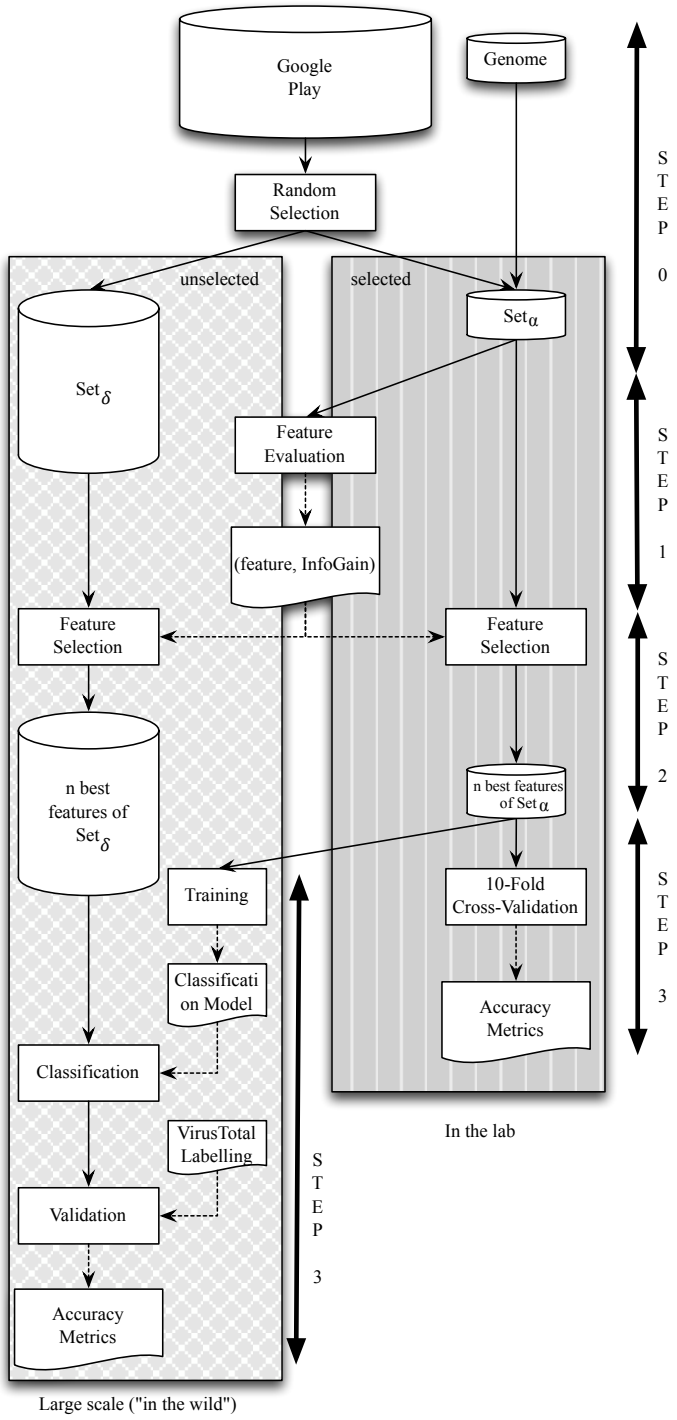


Figure 6.1: The steps in our approach

Step 0: Set composition Our complete dataset contains over 50 000 applications that we divide into two distinct sets, one significantly smaller than the other, for the purpose of assessment. The first set, Set_α, contains all known malware, i.e., all items in the *Genome* dataset. To complete this set, we randomly select a subset of the *Google Play* dataset to add as the goodware portion of the dataset. The second set, Set_δ, is then composed of the remaining subset of the Google Play dataset. Set_δ is always used as a testing set, whereas Set_α can be used as training set (in the wild) or as the entire universe (10-Fold), i.e., testing and training sets combined (cf. Fig. 6.1).

Step 1: Feature Evaluation Once the sets of an experiment are defined, a feature evaluation step is performed to measure the discriminating power of every feature. This measure is computed using the InfoGain Feature evaluation as implemented in the Machine Learning software Weka[5] (Hall et al., 2009).

Step 2: Feature Selection For practical reasons, given the large sizes of the datasets, hence the high number of features to process, we must improve computation efficiency by reducing the number of features. Indeed, reducing the number of features considered for the classification will decrease the working size of the sets, leading to lowered I/O, memory and CPU consumption for the subsequent processing steps. In our approach we only retain, after the evaluation step, the best N features, i.e. those with the highest InfoGain values. The number of features is reduced in both the training set and the testing set. For every built training set, we derived about 2.5 millions features, and over 99% of them had a null (0) InfoGain measure. We thus discard those features whose null discrimination power implies that they are "irrelevant". Previous work has already demonstrated that removing such irrelevant features may, beyond computation efficiency gain, improve classifiers' ability to generalize its model (Tahan et al., 2012), which in turn could lead to a better detection of previously unknown malware.

Step 3: Classification validation scenarios We propose to use two distinct scenarios to validate our malware detection approach.

Validation in the lab Traditionally, machine learning-based approaches are assessed in a cross validation scenario that validates the classification model by assessing how the result will generalize to an independent dataset. To estimate how the prediction model will perform in practice, a cross-validation scenario partitions the sample data into 2 subsets. The first subset is used for learning analysis, i.e., building the model during the training phase. The second subset is used to validate the model. However, to reduce variability of the results, multiple rounds are performed and the results are averaged over the rounds. A well-known type of cross-validation is the 10-Fold cross validation (McLachlan, Do, & Ambroise, 2005) which randomly partitions the sample data into 10 subsamples, 9 of which are used for training and 1 for validation. The process is then repeated with each subsample being used exactly once for validation. This method enables to

[5] http://www.cs.waikato.ac.nz/ml/weka/

consider all elements in the original sample for training but to have each element validated only and exactly once. For assessing our malware detection approach with the 10-Fold cross validation scheme we consider Set_{α}, which was defined in *Step 0*, as the dataset where both training and testing data will be drawn. This dataset contains both malware and goodware. Every Android application of this dataset will then be classified exactly once, allowing us to easily determine the performance of our approach in this setting.

Another common aspect of *in the lab* validation is the size of the dataset, usually a few thousands applications at most as can be seen in table 6.1.

Validation in the wild. Unfortunately, the 10-Fold cross validation scenario as it is described above does not quite capture the real-world settings in which the malware detector is intended to be used. Indeed, by splitting a dataset in 10 parts, 9 of which are used for training, a 10-Fold cross-validation implicitly assumes that 90% of the domain knowledge is known beforehand—a condition that contradicts the very idea of *in the wild*.

A 10-Fold cross-validation experiment only serves to validate that a given classifier performs well in this one set of conditions, and not that its performance can be generalised outside the scope of these datasets. In the wild, the malware detection tool will only know a size-constrained sample of malware. It could also know a few true goodware, the majority of applications being of an unknown class. To detect malware in this last category, the malware detection tool must be able to perform at large.

We perform large-scale experiments where the classification algorithm of our approach is trained on Set_{α}. To investigate the impact of the quality of the training set, we perform two rounds of experiments where the randomly selected "goodware" from the Google Play dataset are alternatively just considered as such, or confirmed and cleaned, as true goodware using antivirus products. The trained classifier obtained is then used to predict the class, either *malware* or *goodware*, of every single application from Set_{δ}. Those predictions are finally compared to our reference malware classification obtained from VirusTotal to assess the performance of the approach in the wild.

6.4.3 Varying & Tuning the Experiments

In this section we succinctly describe the parameters that are used in our experiments to vary and tune the experiments to share insights in the practice of malware detection with machine learning techniques. These parameters were selected in accordance with the research questions outlined previously in Section 6.3.2.

Table 6.1: Recent research in Machine Learning-based Android Malware Detection

Authors	Features	Algorithm	Evaluation	Datasets	Training set	Test Set	Comment
Sahs and Khan (2012)	Permissions, CFG subgraphs	1-class SVM	k-fold	2 081 goodware 91 malware	Subsets of the goodware set	91 malware (and remainder of training set?)	Sahs & Khan approach yielded high recall with low precision. The vast majority of our *in the lab* classifiers yielded both a high recall and a high precision.
Amos, Turner, and White (2013)	Profiling (Dynamic)	RandomForest, C4.5, etc.	10-fold on training set and evaluation on a test set	1 777 Apps	408 goodware 1 330 malware	24 goodware 23 malware	Our closest experiment (goodware/malware ratio: 1/2) yielded dozens of classifiers with equivalent or better performance
Yerima, Sezer, McWilliams, and Muttik (2013)	API calls, external tool execution, permissions (Static)	Bayesian	5-fold	1 000 goodware 1 000 malware	?[1]	?[1]	Our closest *in the lab* experiment (goodware/malware ratio: 1) yielded 74 classifiers with both higher recall and higher precision than Yerima et al.'s best classifier.
Demme et al. (2013)	Performance Counters (Dynamic)	KNN, RandomForest, etc.	?[1]	210 goodware 503 malware	?[1]	?[1]	The majority of our *in the lab* classifiers yielded higher recall and higher precision than Demme et al.'s best classifier
Canfora, Mercaldo, and Visaggio (2013)	SysCalls, Permissions	C4.5, RandomForest, etc.	?[1]	200 goodware 200 malware	?[1]	?[1]	In our closest experiment by dataset size (goodware/malware ratio :1/2), our worst classifier performs better than Canfora et al.'s best classifier. In our closest experiment by goodware/malware ratio (1), the vast majority of our classifier perform better than Canfora et al.'s best classifier.
Wu, Mao, Wei, Lee, and Wu (2012)	Permissions, API Calls, etc.	KNN, NaiveBayes	?[1]	1 500 goodware 238 malware	?[1]	?[1]	More than 100 of our *in the lab* classifiers yielded both a higher recall and a higher precision than their best classifier.

[1]We were unable to infer this information.

6.4.3.1 Goodware/Malware ratio

We see a first parameter in the building of the datasets. Indeed, given that the size of the malware set is fixed and known, what size of goodware should be selected in the very large set of goodware available to yield a *good ratio*? We performed various experiments to analyze the impact of

the potential class imbalance between in the dataset, tuning the ratio value to 1/2, 1, 2 and up to 3, representing respectively 620, 1 247, 2 500 and 3 500 Android applications selected in the goodware set. Having the vast majority of examples from one of the classes, aka class imbalance, is a well-documented threat to Machine Learning performance in general (Van Hulse, Khoshgoftaar, & Napolitano, 2007; He & Garcia, 2009). This threat is even more severe in malware detection because of the relative scarcity of malware in comparison to the number of available benign applications. Yet, surprisingly, the literature of machine learning-based malware detection often eludes this question in experiments (Cf. Section 3.3).

6.4.3.2 Volume of processed features

Feature selection is an important step of the classification model. However, it can bias the output of the classification depending on the threshold that is set for defining *best* features. We investigate the role played by the number of features considered as relevant for our malware detector. To this end, we vary this number for the values of 50, 250, 500, 1 000, 1 500, 5 000.

6.4.3.3 Classification algorithm

Last, as introduced in the description of the classification model, our malware detectors are implemented using 4 different algorithms which are well-known in the community of machine learning. For all algorithms, we have used existing implementations in Weka, namely RandomForest, J48, JRip and LibSVM, that were already referred to in the literature. In all of our experiments, these algorithms are used with the default parameters set by the Weka framework.

Overall, since the selection of Goodware performed in Step 1 of the classification is performed randomly, we reduce variability of the results by repeating 10 times each experiment with a given triplet of parameter values. In total, 4 (values for number of Goodware) $\times 6$ (values for number of features) $\times 4$ (number of algorithms) $\times 10 = 960$ runs were processed for our experiments. The entire process took over thirty (30) CPU-days to complete.

6.5 Assessment

In this section we present an extensive assessment of our machine learning-based malware detection approach. We first validate the approach using a typical *in the lab* validation scenario, while

discussing the impact of the different parameters that are involved in the process. Second, we compare the performance of our malware detector with approaches in the literature to highlight the relevance of our feature set. However, we take the experiments further to investigate the capability of malware detectors to scale in the wild.

6.5.1 Evaluation *in the lab*

We run 960 10-Fold cross validation experiments with all combinations of parameter values to assess the performance of our malware detection approach. Because in each experiment the goodware set is varied, computed features vary, and thus the classification model leads to distinct classifiers. The validation thus assesses altogether the 960 classifiers that were built in the experiments. Figure 6.2 depicts the distribution of *precision*, *recall* and *F-measure* that the validation tests have yielded. In each boxplot diagram presented, whiskers go from the minimum value recorded to the maximum value. The box itself is built as follows: the bottom line of the box represents the 25^{th} percentile; the top of the box represents the 75^{th} percentile; the line inside the box represents the median value.

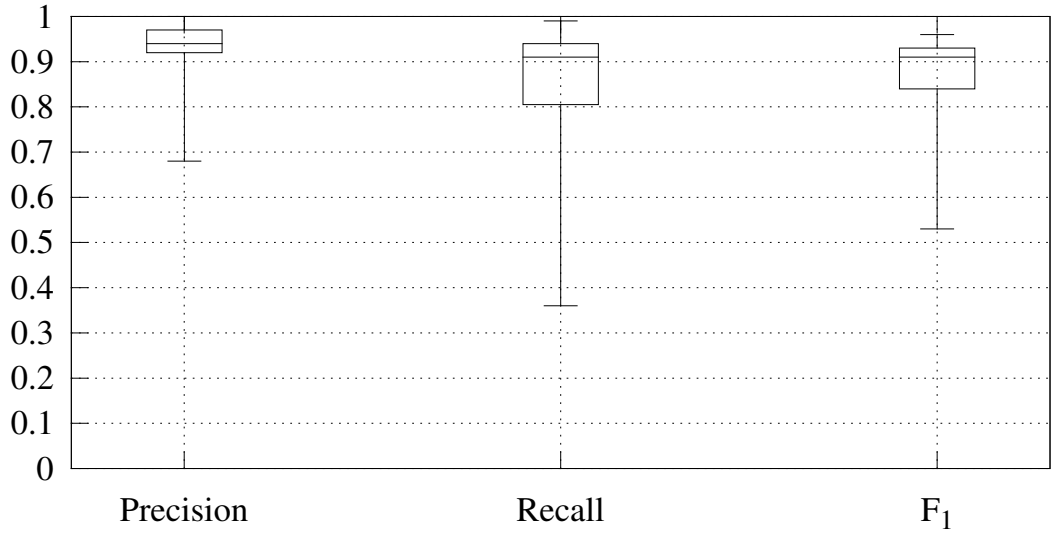


Figure 6.2: Distribution of precision, recall and F-measure for the *malware* class yielded in all 960 *in the lab* experiments

Overall, the results indicate that the vast majority of our 960 built classifiers exhibit a very high precision rate with a median value of 0.94. The median value of recall is recorded at 0.91, meaning that half of the classifiers have recall values that are equal or higher to 0.91. Although recall values are lower than precision values, a large portion of the built classifiers exhibit a high

recall rate. Given the precision and pecall rates, the F-measure values obtained are globally high, going from 0.53 to 0.96, with a median value of 0.91.

6.5.1.1 Impact of class imbalance

We now investigate in detail how class imbalance in the constructed dataset threatens the performance of machine learning-based malware detectors, and thus, how a collection of unrealistic datasets can bias validation results. To this end, as announced in Section 6.4.3, we perform *in the lab* experiments using datasets where the goodware/malware ratio is varied between 1/2 and 3. All other parameters are varied across all their value ranges.

Figure 6.3 shows that when the goodware/malware ratio is increasing in favor of goodware, the precision of malware detectors increases, while its recall decreases. The increase of the precision can be attributed to the fact that the classification model has a better view of the universe and can discriminate more accurately malware against goodware. However, at the same time, the classifiers can no longer recognize all malware since most will be more similar to some of the too many goodware. This drop in recall rate is so marked that the overall performance, measured with F-measure, decreases as revealed by the boxplots of Figure 6.3. This observation is of particular importance in the field of malware detection since, in real-world scenarios, there is much more goodware than malware.

> **RQ1:** *The performance of the machine learning-based malware detector decreases when there are fewer malware than goodware in the training dataset.*

6.5.1.2 Sensitivity to the volume of relevant features

We survey the effect that an implementation choice on the number of relevant features to retain for classification can have on the performance of the malware detector. In each experiment, about 2.5 millions distinct features are generated, most of which are evaluated to being completely irrelevant. Using the remaining features, we successively select between 50 and 5 000 to use as relevant features for the classifiers. Figure 6.4 shows that the overall performance, measured with F-measure, is improving with the number of features retained. However the figure also shows that over a certain threshold number, about 1 000, of features, the median value of F-measure is no longer affected. The improvement is thus confined at the upper level.

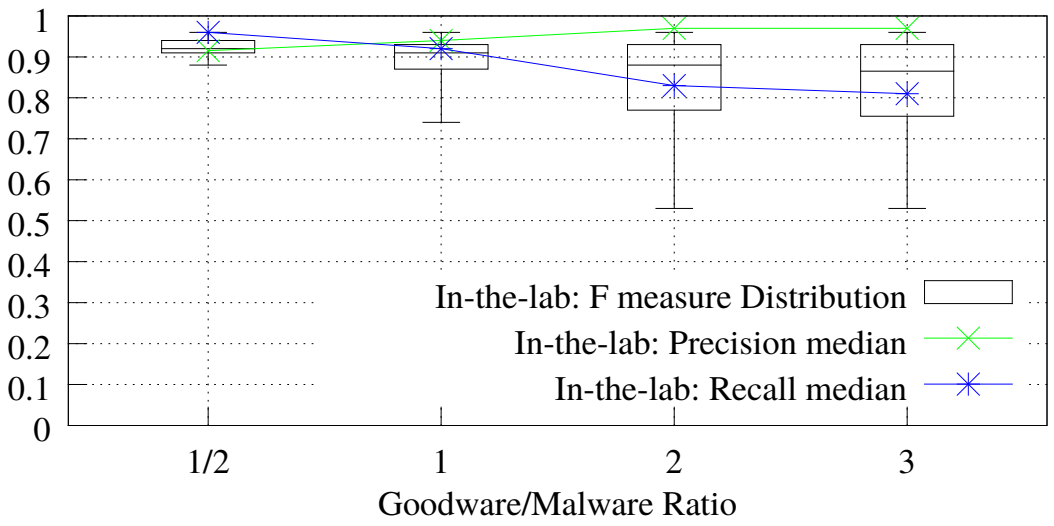


Figure 6.3: Distribution of F-measure and evolution of precision and recall for various goodware/malware ratio values

RQ2: *The more features are considered for the training phase, the better the performance of the malware detector.*

6.5.1.3 Effect of classification algorithm

Finally, we investigate the role played by the classification algorithm in the variation of performance between classifiers. To that end we compare the performance of classifiers after regrouping them by the underlying algorithm. Figure 6.5 represents the distribution of F-measure for the 4 algorithms that are used in our experiments. RandomForest, the RIPPER rule-learning algorithm, and *C4.5* exhibit high F-measure rates. SVM on the other hand provides results with a wider distribution and an overall lower F-measure.

Figure 6.6 plots the values of precision and recall for all classifiers built when using each algorithm. We note that SVM leads to numerous classifiers with precision values close to 1, but that present lower recall rates than the other algorithms. Although SVM yields the best classifiers—the top 66 classifiers with highest precision and the top 42 with highest recall are based on SVM—it tends in our approach to yield few classifiers that have both good precision and good recall.

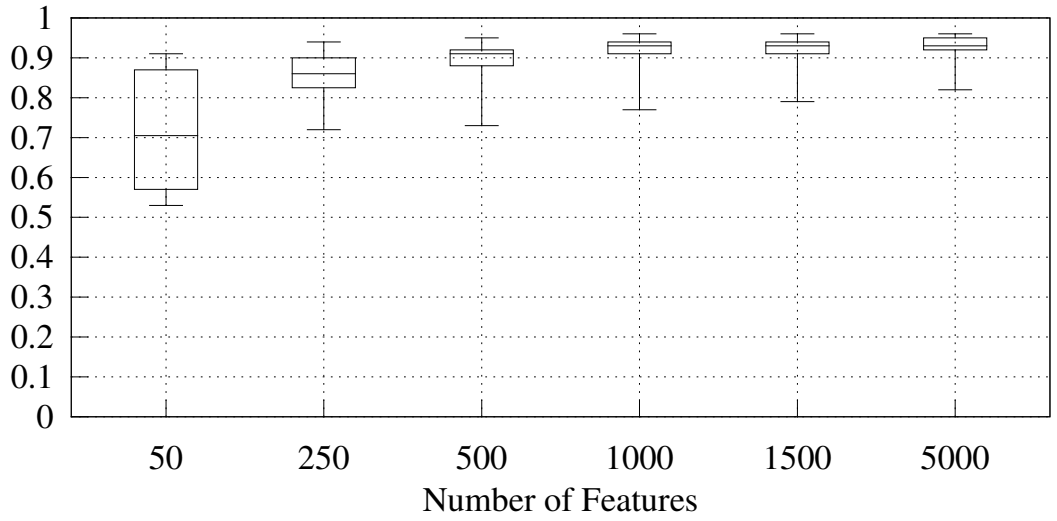


Figure 6.4: Distribution of F-measure for different volumes of the set of considered relevant features

RQ3: *Four common classifications algorithms have led to similar performance with our feature set, suggesting that the approach is not tailored to a specific algorithm.*

6.5.2 Comparison with Previous work

Table 6.1 in appendix summarizes a number of state-of-the-art machine learning-based malware detection approaches for the Android platform. We indicate the features that are used, the type of validation that were performed in the paper, the sizes and composition of the training set, the size of the testing set, if known, and an overall performance comparison with our approach. Overall, we note that our cross validation experiments have yielded at worst similar performance than state-the-art approaches, and at best, our worst classifiers perform better than classifiers of approaches in the literature. All comparisons were done on equivalent experiments, i.e., with similar training and testing sets, and the same classification algorithms whenever possible.

We provide this comparison to provide a settings for a stronger, and more general, discussion on the scope of 10-Fold cross validation for approaches that are meant to be applied on datasets in the wild.

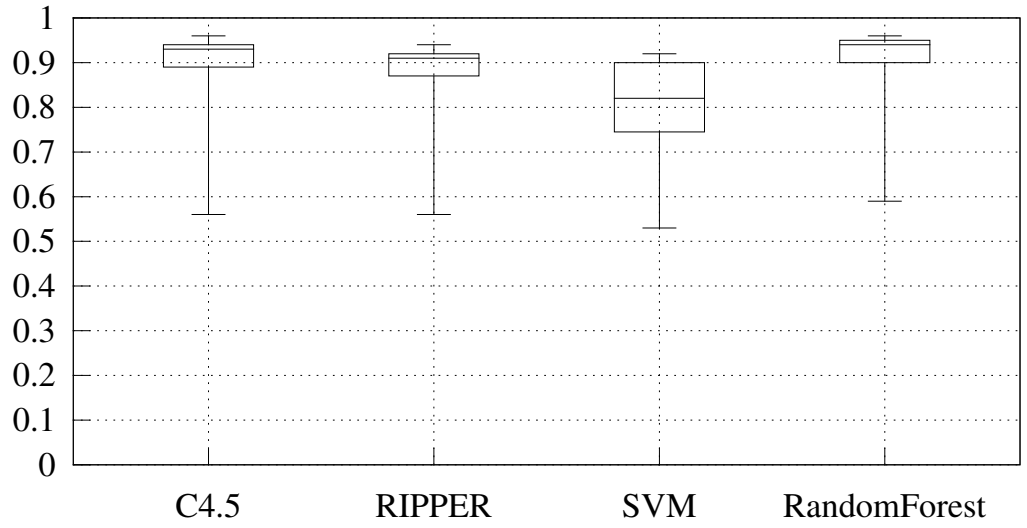


Figure 6.5: Distribution of F-measure for 4 different classification algorithms

Finding: *Our classifiers, when built with similar parameters than existing approaches, and evaluated* in the lab*, are highly performant.*

6.5.3 Evaluation in the wild

Beyond simply demonstrating the performance of our malware detection approach using cross-validation, we explore in this section its performance in the wild. We perform large-scale experiments on sizes of datasets that are unusually large for the literature of malware detection, but that better reflect realistic use-cases. Two points should be highlighted:

- 10-Fold cross-validation assesses the performance of a classifier by considering 90% of the dataset for training, thus supposing a prior knowledge of the malware class of each application in 90% of the dataset. Real-world datasets of applications however present a contrasting specificity: the known malware set is limited and is insignificant compared to the rest (i.e. goodware + unknown malware).

- Performance assessment of malware detectors should be carefully performed so as to expose the scope in which they can be of use in real-world settings. Thus, large-scale experiments with varying parameters can help refine a methodology for using, in realistic settings,

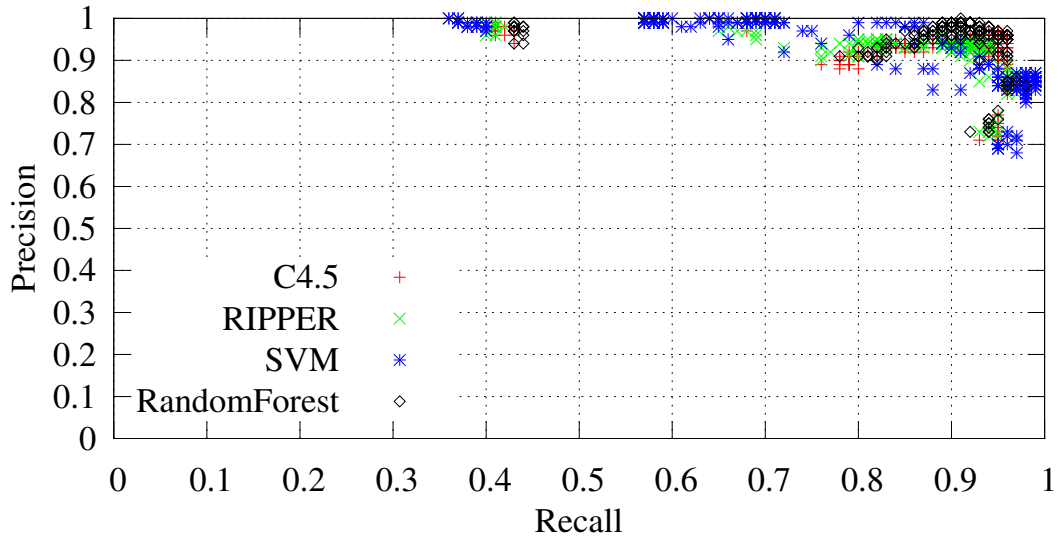


Figure 6.6: Precision and recall values yielded by all classifiers for the 4 different classification algorithms

a malware detection approach that was shown successful with 10-Fold cross validation on controlled datasets.

The experimental protocol used in this evaluation is similar to that used in the validation experiments of Section 6.5.1, except that we do not perform 10-Fold cross validation. Instead, we use our entire Training data, i.e., the entire set of known malware + a randomly selected subset of the goodware, to build the classification model (cf. Figure 6.1). By varying the different parameters explicited in Section 6.4.3, we obtain again 960 classifiers that will be used to test the large remaining set of goodware containing from 48 422 to 51 302 applications. Each experiment with a specific set of parameters is repeated 10 times to stabilize the results. Indeed, since step 0 of our experimental setup randomly selects parts of the training dataset, repeating experiments ten times, each with a different training-set prevents the results from being biased by the possibility that the randomly selected training set is particularly *good* or particularly *bad*.

The predictions of the malware detector are then checked against the independent reference classification (cf. Section 6.3.3).

Figure 6.7 illustrates the distribution of precision, recall and F-measure values for the 960 classifiers that were built during the large-scale experiments. Overall, the classifiers exhibit a very low precision rate with a median value of 0.11. We have enumerated 13 classifiers with the highest precision value of 1. However, these only classified between 5 and 7 applications, thus yielding an exceedingly low recall rate. Also, most of the 960 classifiers have a recall value close to 0. Even the unique classifier which provided a 0.45 recall value had to classify half of the dataset as malware. Finally, with a low precision and an even lower recall, the global performance of

the classifiers severely drops in large-scale experiments, with a majority of classifiers yielding a F-measure value close to 0.

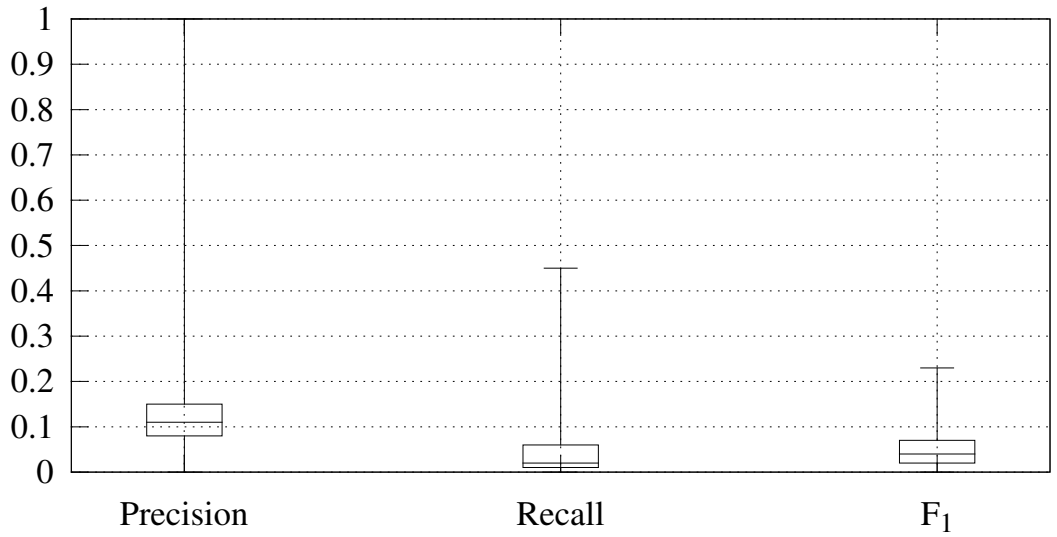


Figure 6.7: Distribution of precision, recall and F-measure values in *in-the-wild* experiments

Figure 6.8 shows that when the ratio of goodware/malware in the training set is balanced in favor of the goodware set in training data, the precision rates increase slightly while recall values decrease rapidly. This figure shows that a class imbalance in favor of the goodware set leads to an overall performance drop, with the F-measure values closer to 0.

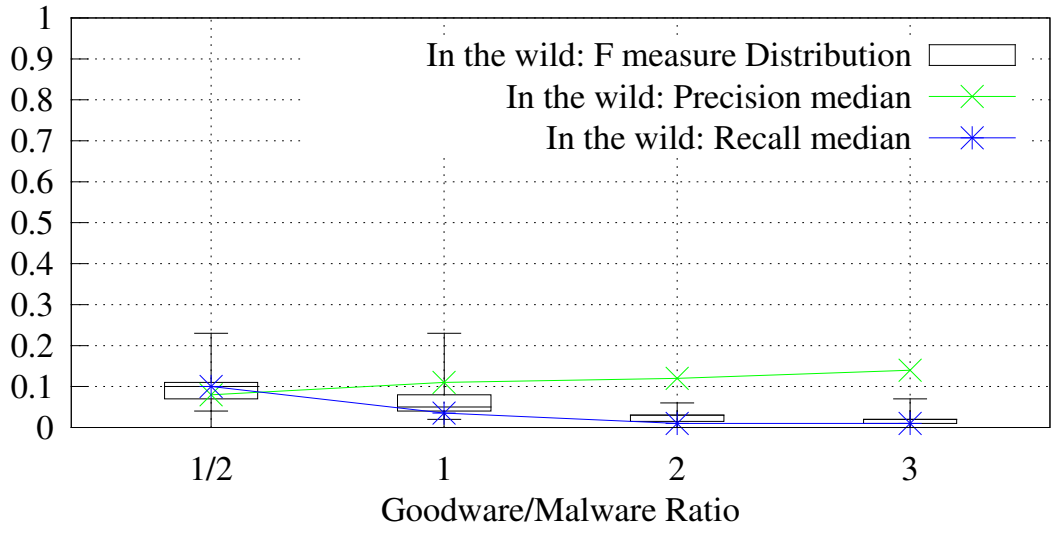


Figure 6.8: Distribution of F-measure and evolution of precision and recall for various goodware/malware ratio values in *in-the-wild* experiments

Again, as in the case of *in the lab* experiments, we investigate the sensitivity of the malware detector to the volume of relevant features. Figure 6.9, which depicts the distribution of F-measure values for different experiments with varied number of features that are kept as relevant, shows that, in the wild, their impact is not significant. Indeed, aside from the first boxplot for a really small number, 50, of features, all other boxplot show a compact distribution with similarly low median values.

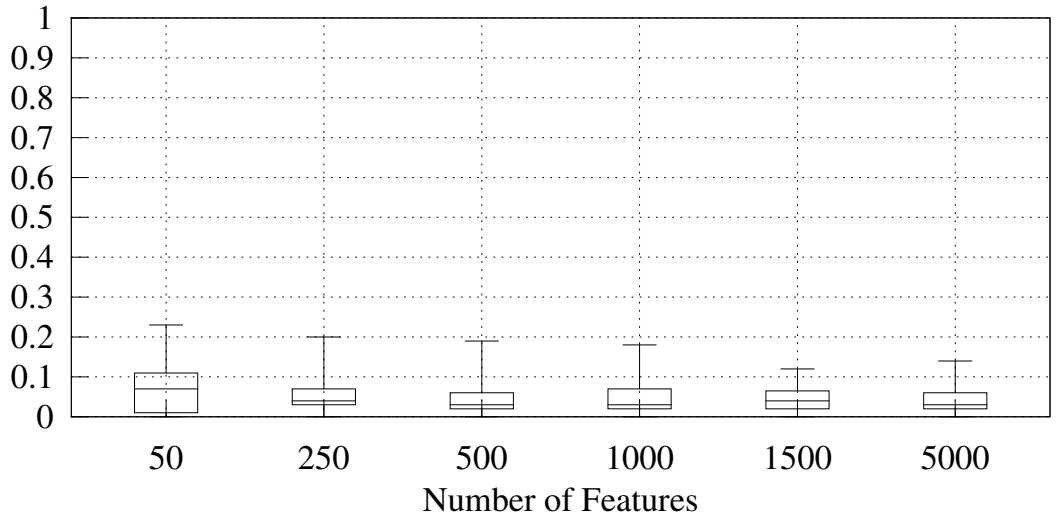


Figure 6.9: Distribution of F-measure for different volumes of the set of considered relevant features in *in-the-wild* experiments

Finally, Figure 6.10 presents the distribution of F-measure for classifiers built based on the four different classification algorithms used in our experiments. The distributions reveal that no algorithm significantly outperforms the others for our experiments in the wild.

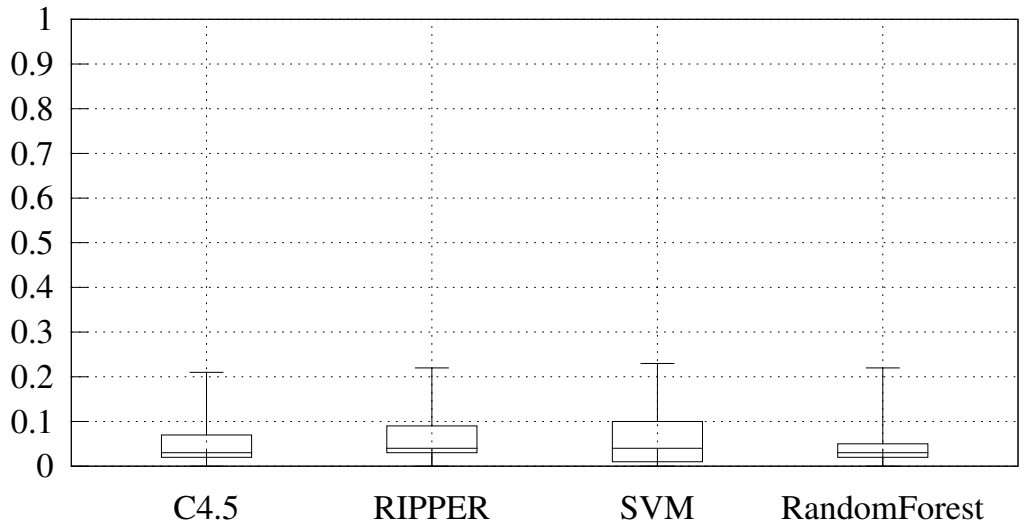


Figure 6.10: Distribution of F-measure for different algorithms in *in-the-wild* experiments

Summary: *In the wild, experiments have revealed a poor overall performance of the malware detectors. Variations of goodware/malware ratio and classification algorithms yield the same evolutions as for* in the lab *experiments. In contrast, increase in the volume of features lead to a drop in performance during large-scale experiments.*

6.6 Discussion

In the lab experiments with the 960 different built classifiers have demonstrated that our malware detection approach performs well in comparison with existing approaches in the literature. However applying those classifiers to detect malware in very large datasets have yielded very low performance. Figure 6.11 illustrates the contrasting F-measure median values for both experimental scenarios with varying number of features.

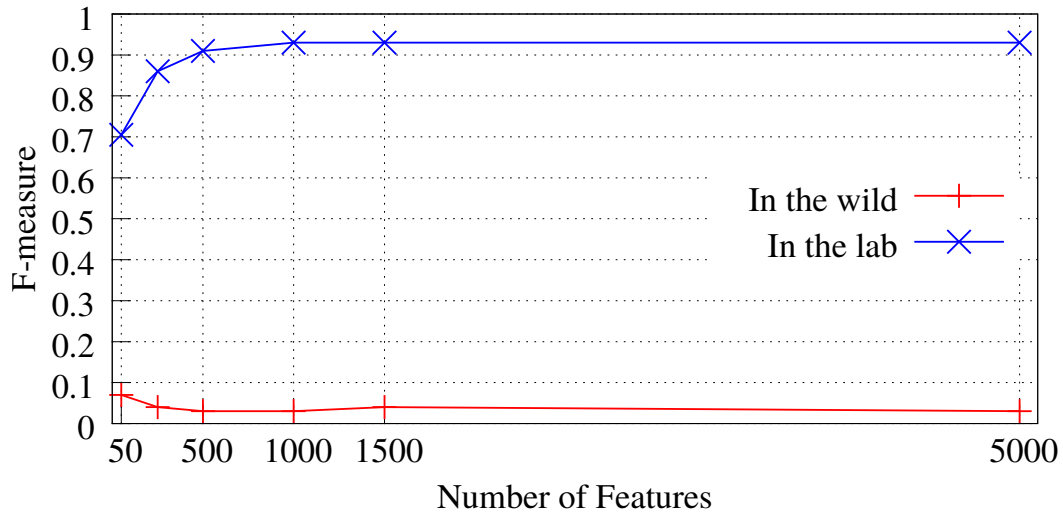


Figure 6.11: Comparison of F-measure median values

We now enumerate a few points that are relevant to discuss the performance of Malware classifiers in the wild.

6.6.1 Size of training sets:

Given the importance of the training phase, it could be argued that the size of training set that we have used in large-scale experiments are too small compared to the size of the testing set. Nonetheless, the gap between these sizes is in respect with real-world scenarios as discussed in Section 6.2. Furthermore, our experiments, illustrated in Figure 6.8, have shown that the Recall rates actually decreases when the size of training set increases.

6.6.2 Quality of training sets:

The poor performance of classifiers during experiments in the wild could be attributed to some potential noise in the "goodware" set collected from Google Play; i.e., some goodware in this set are actually unknown malware whose features are biasing the classification model. Indeed, according to detection reports from VirusTotal, 16% of the applications obtained from Google Play are malware. We have then run experiments where the training data contained alternatively a goodware set that were uncleaned and a goodware set that were cleaned with antivirus products. Figure 6.12 shows the slight improvement that cleaned dataset provides. Nonetheless, the global performance remains significantly low. Furthermore, since, to the best of our knowledge, there is no publicly available collection of known goodware that one can rely upon, a good classifier should perform relatively well even in presence of noisy training datasets.

RQ4: *The machine learning-based malware detector is sensitive to the quality of training data. A cleaned goodware set positively impacts overall performance.*

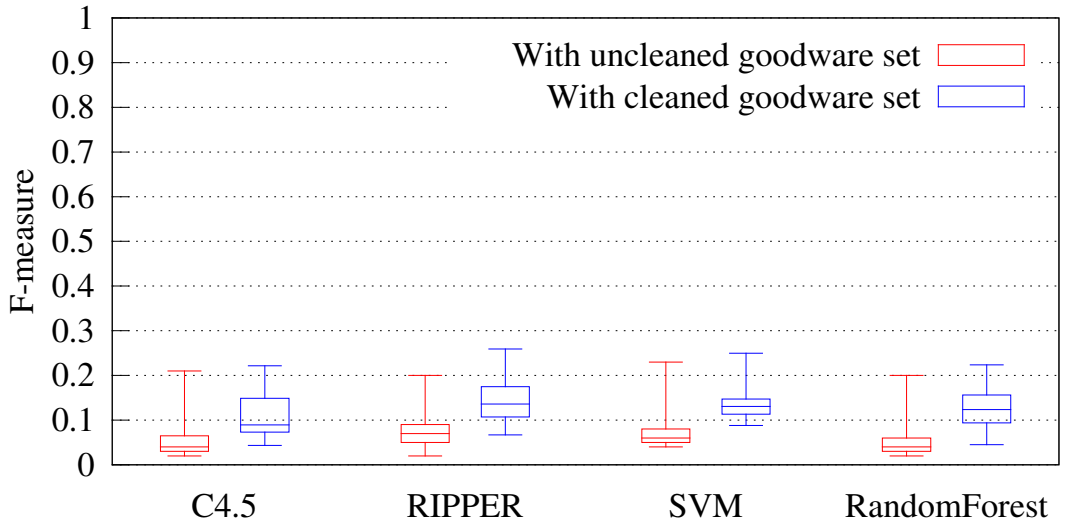


Figure 6.12: Distribution of F-measure values with cleaned and uncleaned goodware sets for experiments *in the wild*

6.7 Threats to Validity

Our study presents a number of threats to validity that we discuss in the following to highlight their potential impact on our findings and the measures we have taken to mitigate their bias.

6.7.1 External Validity

Datasets representativity: During collection of datasets from Google Play, we did not consider downloading any paid application. However, free applications account for the majority of Android applications available (AppBrain, 2013a) and appear to be the most affected by malware.

Furthermore, the malware from the Genome dataset that we have used may not be representative enough of the malware corpus available in Google Play. However, to the best of our knowledge, this is the most comprehensive collection of Android malware available to researchers in the Security and Privacy field. Besides, malware representativity is hard to define in practice, since it would require that one knows beforehand all malware that are being looked for.

Google's own malware detector: In February 2012, Google announced (Google, 2012) they were using *Bouncer*, their own Android malware detector, to prevent malicious applications to reach the official Google Play market. While Bouncer still allows many malware to enter Google Play (Allix, Jérome, et al., 2014), it may bias our dataset collection.

Since both our *in the lab* and *in the wild* experiments used apps collected from Google Play, both validation scenarios should be affected by this bias. Bouncer therefore cannot play a significant role in the performance gap we observed. However, if Bouncer had a negative impact on Android malware detectors, our results show that this impact would be marginal *in the lab*, but significant *in the wild*, hence highlighting the importance of *in the wild* experiments.

6.7.2 Construct Validity

Labeling methods: In our experiments, two different reference classification sources were used as ground truth: *in the lab* experiments were based on the Genome project classification alone while *in the wild* experiments used the Genome project for training and were tested against VirusTotal classification. Although we verified beforehand that **every app from the Genome project is classified as malware by VirusTotal**, the use of two different labeling sources could be one possible explanation for the differences in accuracy we found when comparing *in the lab* with *in the wild* experiments. To investigate this hypothesis, we performed the same experiments again, this time using only VirusTotal for both training and testing. As can be seen on Fig 6.13, using a single, coherent reference classification does not result in significantly different results. Hence, the performance gap between *in the lab* and *in the wild* experiments cannot be explained by our usage of labelling sources.

Exhaustiveness of classification algorithms: Machine-learning algorithms perform differently depending on the context. It is thus possible that the four well-known algorithms that we have selected were used in this study outside of their comfort zone. Nonetheless, we note that 3 very distinct algorithms exhibited similar patterns, suggesting that our findings are not specific to a particular type of classification algorithm.

Relevance of feature set: Our experiments were performed with the same type of features, which are based on basic blocks of CFGs. Possibly, this particular feature set is incompatible with experiments in the wild. However, we have not found in the state-of-the-art literature evidence suggesting that other feature sets with high performance in *in the lab* validation actually perform well in large-scale experiments as well.

Limited experiments with 2-grams extracted from raw bytecode, resulted in the same performance gap between *in the lab* and *in the wild* validation scenarios.

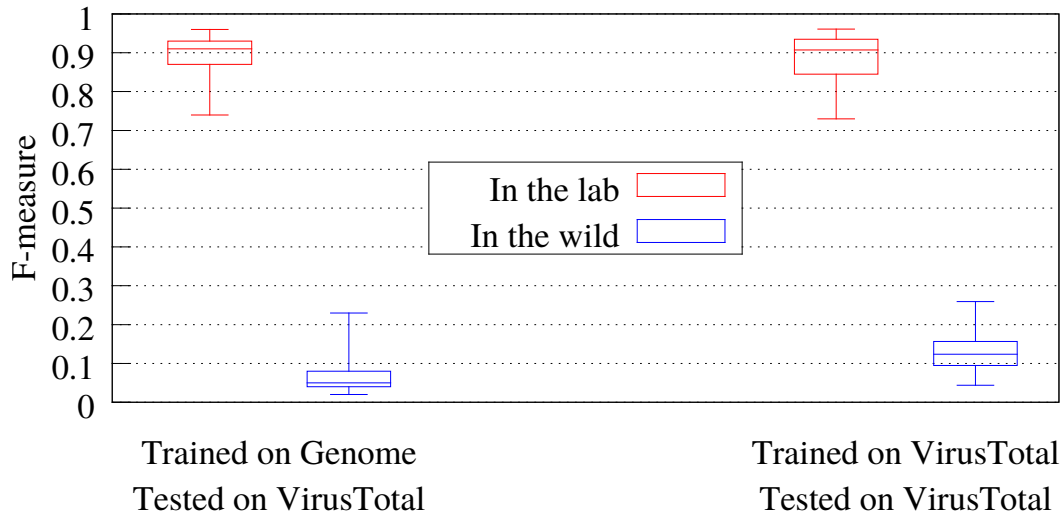


Figure 6.13: Distribution of F-measure for different classification references usages

Furthermore, we note that if our feature-set was deemed unsound, or unsuitable for this study, this would actually strengthen our argument. Indeed, it would demonstrate that even an unsound feature-set can lead to high-performance *in the lab*, or in other words, that high performance *in the lab* is not even a valid indicator of soundness for a feature-set.

6.7.3 Internal Validity

Composition of training and testing sets: The size of training sets and the ratio between goodware and malware sets take various values that appear to be unjustified since, to the best of our knowledge, no survey has determined their appropriate values for malware detection. However, we have ensured that the sizes that are used in our study are comparable to other research work, and that they are representative of the data available to the research community.

6.7.4 Other Threats

Specificity of Findings to the Android platform: Experiments in this study focused on Android applications. We have not studied malware detection for other Operating Systems. Although our feature set does not take into account any specificities, such as Android Permissions scheme, we

cannot rule out that the gap between *in the lab* and *in the wild* scenarios could be narrower in other platforms.

6.8 Conclusion

We have discussed in this chapter the validation of machine-learning malware detection with *in the lab* and *in the wild* scenarios. A first contribution of our work is a Feature set for building classifiers that yield high performance measures in *in the lab* evaluation scenarios and in comparison with state-of-the-art approaches. Beyond this evaluation, however, we have assessed the actual ability of our classifiers to detect Malware in a significantly large dataset. The recorded poor performance has provided us with new insights as to the limits to which an *in the lab* validation scheme is a reliable indicator for real-world malware detectors. We have thus identified several parameters that are likely to impact the performance of Malware Detectors. Finally, we make available to the research community all our datasets to improve the research on Android malware detection.

Our Argument. By presenting here an approach that exhibits high performance *in the lab* and yet has little practical usefulness in the wild, we demonstrated that there exists at least one approach for which this performance gap exists. While this chapter cannot demonstrate that the same gap exists for other published approaches, we claim that until those approaches are tested in the wild, they cannot be supposed to represent a significant improvement to the malware detection domain.

We also showed here that this issue of validation scenario is not merely a minor bias in experimental results: *in the lab* results are not a slightly optimistic version of results *in the wild*. Instead, they can be vastly different and tell widely different stories.

Hence, evaluating malware detector *in the wild*, with a sound empirical methodology is of the utmost importance. In other words, we call for the Machine Learning-based malware detection community to devise and agree on what would be sound, in-depth and meaningful validation scenarios.

In future work, we plan to investigate the reasons of the observed performance gap, and to formalise a methodology for sound, extensive, reliable and reproducible empirical evaluation of malware detectors.

Chapter 7

History Matters

Contents

This chapter is based on work published in Allix, K., Bissyandé, T., Klein, J., & Le Traon, Y. (2015). Are your training datasets yet relevant? an investigation into the importance of timeline in machine learning-based malware detection. In F. Piessens, J. Caballero, & N. Bielova (Eds.), *Engineering secure software and systems* (Vol. 8978, pp. 51–67). Lecture Notes in Computer Science. Springer International Publishing. doi:10.1007/978-3-319-15618-7_5

7.1 Introduction

Malware detection is a challenging endeavor in mobile computing, where thousands of applications are uploaded everyday on application markets (AppBrain, 2013b) and often made available for free to end-users. Market maintainers then require efficient techniques and tools to continuously analyze, detect and triage malicious applications in order to keep the market as clean as possible and maintain user confidence. For example, Google has put in place a number of tools and processes in the Google Play official market for Android applications. However, using antivirus software on large datasets from Google reveals that hundreds of suspicious apps are still distributed incognito through this market (Cf. chapter 5).

Unfortunately, malware pose various threats that cannot be ignored by users, developers and retailers. These threats range from simple user tracking and leakage of personal information (Enck et al., 2011), to unwarranted premium-rate subscription of SMS services, advanced fraud, and even damaging participation to botnets (Pieterse & Olivier, 2012). To address such threats, researchers and practitioners increasingly turn to new techniques that have been assessed in the literature for malware detection in the wild. Research work have indeed yielded promising approaches for malware detection. A comprehensive survey of various techniques can be found in Idika and Mathur (2007). Approaches for large-scale detection are often based on Machine learning techniques, which allow to sift through large sets of applications to detect anomalies based on measures of similarity of features (Arp et al., 2014; Chau, Nachenberg, Wilhelm, Wright, & Faloutsos, 2010; Boshmaf et al., 2012; Su, Chuah, & Tan, 2012; Henchiri & Japkowicz, 2006; Kolter & Maloof, 2006; Zhang et al., 2007; Sahs & Khan, 2012; Perdisci et al., 2008a).

To assess malware detection in the wild, the literature resorts to the 10-Fold Cross validation scheme with datasets that we claim are biased and yield biased results. Indeed, various aspects of construction of training datasets are usually overlooked. Among such aspects is the *history aspect* which assumes that the training dataset, which is used for building classifiers, and the test dataset, which is used to assess the performance of the technique, should be *historically coherent*: the former must be historically anterior to the latter. This aspect is indeed a highly relevant constraint for real-world use cases and we feel that evaluation and practical use of state-of-the-art malware detection approaches must follow a process that mimics the history of creation/arrival of applications in markets as well as the history of appearance of malware: *detecting malware*

before they are publicly distributed in markets is probably more useful than identifying them several months after they have been made available.

Nevertheless, in the state-of-the-art literature, the datasets of evaluation are borrowed from well-known labelled repositories of apps, such as the Genome project, or constructed randomly, using market-downloaded apps, with the help of antivirus products. However, the history of creation of the various apps that form the datasets are rarely, if ever, considered, leading to situations where **some items in the training datasets are "*from the future*", i.e., posterior, in the timeline, to items in the tested dataset**. Thus, different research questions are systematically eluded in the discussion of malware detector performance:

> **RQ-1.** Is a randomly sampled training dataset equivalent to a dataset that is historically coherent to the test dataset?

> **RQ-2.** What is the impact of using malware knowledge "from the future" to detect malware in the present?

> **RQ-3.** How can the potential existence of families of malware impact the features that are considered by machine learning classifiers?

> **RQ-4.** How *fresh* must be the apps from the training dataset to yield the best classification results?

> **RQ-5.** Is it sound/wise to account for all known malware to build a training dataset?

This Chapter. We propose in this chapter to investigate the effect of ignoring/considering historical coherence in the selection of training and test datasets for malware detection processes that are built on top of Machine learning techniques. Indeed we note from literature reviews that most authors do not take this into account. Our ultimate aim is thus to provide insights for building approaches that are consistent with the practice of application –including malware– development and registration into markets. To this end, we have devised several typical machine learning classifiers and built a set of features which are textual representations of basic blocks extracted from the Control-Flow Graph of applications' byte-code. Our experiments are also based on a sizeable dataset of about 200 000 Android applications collected from sources that are used by authors of contributions on machine learning-based malware detection.

The contributions of this chapter are:

- We propose a thorough study of the history aspect in the selection of training datasets. Our discussions highlight different biases that may be introduced if this aspect is ignored or misused.
- Through extensive experiments with tens of thousands of Android apps, we show the variations that the choice of datasets age can have on the malware detection output. To the best of our knowledge, we are the first to raise this issue and to evaluate its importance in practice.
- We confirm, or show how our experiments support, various intuitions on Android malware, including the existence of so-called lineages.
- Finally, based on our findings, we discuss (1) the assessment protocols of machine learning-based malware detection techniques, and (2) the design of datasets for training real-world malware detectors.

The remainder of this chapter is organized as follows. Section 7.2 provides some background on machine learning-based malware detection and highlights the associated assumptions on dataset constructions. We also briefly describe our own example of machine-learning based malware detection. Section 7.3 describes the experiments that we have carried out to answer the research questions, and presents the take-home messages derived from our empirical study. We propose a final discussion on our findings in Section 7.4 and conclude in Section 7.5.

7.2 Preliminaries

The Android mobile platform has now become the most popular platform with estimated hundreds of thousands of apps in the official Google Play market alone and downloads in excess of billions. Unfortunately, as this popularity has been growing, so is malicious software, i.e., malware, targeting this platform. Studies have shown that, on average, Android malware remain unnoticed up to 3 months before a security researcher stumbles on it Apvrille and Strazzere, 2012, leaving users vulnerable in the mean time. Security researchers are constantly working to propose new malware detection techniques, including machine learning-based approaches, to reduce this 3-months gap.

7.2.1 Machine Learning: Features & Algorithms:

As summarized by Alpaydin, "Machine Learning is programming computers to optimize a performance criterion using example data or past experience" Alpaydin, 2010. A common method of

learning is known as *supervised* learning, a scheme where the computer is helped through a first step of *training*. The training data consists of Feature Vectors, each associated with a label, e.g., in our case, apps that are already known to be malicious (*malware* class) or benign (*goodware* class). After a run of the learning algorithm, the output is compared to the target output and learning parameters may be corrected according to the magnitude of the error. Consequently, to perform a learning that will allow a *classification* of apps into the malware and goodware classes, the approach must define a correlation measure and a discriminative function. The literature of Android malware detection includes diverse examples of features, such as n-grams of bytecode, API usages, application permission uses, etc. There also exist a variety of classification algorithms, including Support Vector Machine (SVM) Cortes and Vapnik, 1995, the RandomForest ensemble decision-trees algorithm Breiman, 2001, the RIPPER rule-learning algorithm Cohen, 1995 and the tree-based *C4.5* algorithm Quinlan, 1993. In our work, because we focus exclusively on the history aspect, we constrain all aforementioned variables to values that are widely used in the literature, or based on our own experiments which have allowed us to select the most appropriate settings. Furthermore, it is noteworthy that we do not aim for absolute performance, but rather measure performance delta between several approaches of constructing training datasets.

7.2.2 Working Example

We now provide details on the machine-learning approach that will be used as a working example to investigate the importance of history in the selection of training and test datasets. Practically, to obtain the features for our machine-learning processes, we perform static analysis of Android applications' bytecode to extract an abstract representation of the program's control-flow graph (CFG). We obtain a CFG that is expressed as character strings using a method devised by Pouik *et al.* in their work on establishing similarity between Android applications Pouik and G0rfi3ld, 2012, and that is based on a grammar proposed by Cesare and Xiang Cesare and Xiang, 2010. The string representation of a CFG is an abstraction of the application's code; it retains information about the *structure* of the code, but discards low-level details such as variable names or register numbers. This property is desirable in the context of malware detection as two variants of a malware may share the same abstract CFG while having different bytecode. Given an application's abstract CFG, we collect all basic blocks that compose it and refer to them as the features of the application. A basic block is a sequence of instructions in the CFG with only one entry point and one exit point. It thus represents the smallest piece of the program that is always executed altogether. By learning from the training dataset, it is possible to expose, if any, the basic blocks that appear statistically more in malware.

The basic block representation used in our approach is a high-level abstraction of the atomic parts of an Android application. A more complete description of this feature set can be found in Allix, Bissyandé, Jerome, et al., 2014. For reproducibility purposes, and to allow the research commu-

nity to build on our experience, the data we used (full feature matrix and labels) is available on request.

7.2.2.1 Methodology

This study is carried out as a large scale experiment that aims at investigating the extent of the relevance of history in assessing machine learning-based malware detection. This study is important for paving the road to a true success story of trending approaches to Android malware detection. To this end, our work must rely on an extensive dataset that is representative of real-world Android apps and of datasets used in the state-of-the-art literature.

7.2.2.2 Dataset

To perform this study we collect a large dataset of android apps from various markets: 78,460 (38.04%) apps from `Google Play`, 72,093 (34.96%) from `appchina`, and 55,685 (27.00%) from `Other markets`[1]. A large majority of our dataset comes from `Google Play`, the official market, and `appchina`.

An Android application is distributed as an `.apk` file which is actually a ZIP archive containing all the resources an application needs to run, such as the application binary code and images. An interesting side-effect of this package format is that all the files that makes an application go from the developer's computer to end-users' devices without any modification. In particular, all metadata of the files contained in the .apk package, such as the last modification date, are preserved. All bytecode, representing the application binary code, is assembled into a *classes.dex* file that is produced at packaging-time. Thus the last modification date of this file represents the packaging time. In the remainder of this paper, packaging date and compilation date will refer to this date.

To infer the historical distribution of the dataset applications, we rely on compilation date at which the Dalvik[2] bytecode ($classes.dex$ file) was produced. We then sent all the app packages to be scanned by virus scanners hosted by VirusTotal [3] . VirusTotal is a web portal which hosts about 40 products from renown antivirus vendors, including McAfee®, Symantec® or Avast®. In this study, an application is labelled as malware if at least one scanner flags it as such.

[1] *Other markets* include anzhi, 1mobile, fdroid, genome, etc.
[2] Dalvik is the virtual machine running Android apps.
[3] https://www.virustotal.com

7.2.2.3 Machine learning Parameters

In all our experiments, we have used the parameters that provided the best results in a previous large-scale study Allix, Bissyandé, Jerome, et al., 2014. Thus, we fixed the number of features to 5,000 and selected the 5,000 features with highest Information Gain values as measured on the training sets. The RandomForest algorithm, as implemented in the Weka[4] Framework, was used for all our experiments.

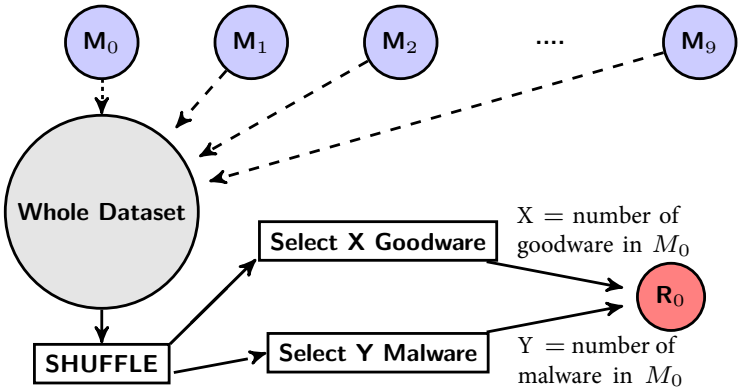


Figure 7.1: Process of constructing a random training dataset R_0 for comparison with the training dataset constituted of all data from month M_0

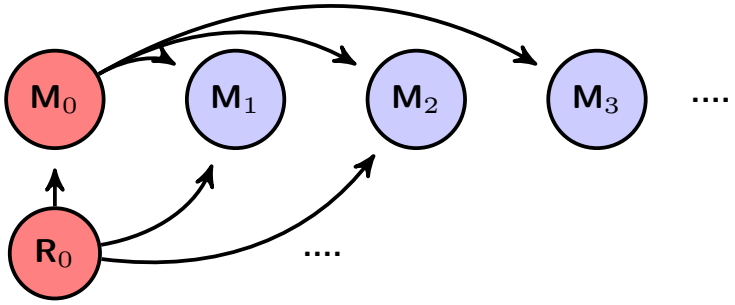


Figure 7.2: Classification process: the training dataset is either the dataset of a given month (e.g., M_0) or a random dataset constructing as in Figure 7.1

[4] http://www.cs.waikato.ac.nz/ml/weka/

7.3 Experimental Findings

In this section, we report on the experiments that we have conducted, and highlight the findings. First we discuss to what extent it is important that datasets remain historically coherent, as opposed to being selected at random (cf. Section 7.3.1). This discussion is based on qualitative aspects as well as quantitative evaluation. Second, we conduct experiments that attempt to provide a hint to the existence of lineages in Android malware in Section 7.3.2. Subsequently, we investigate in Section 7.3.3 the bias in training with new data for testing with old data, and inversely. Finally, we investigate the limitations of a naive approach which would consist in accumulating information on malware samples as time goes, in the hope of being more inclusive in the detection of malware in the future (cf. Section 7.3.4).

7.3.1 History-aware Construction of datasets

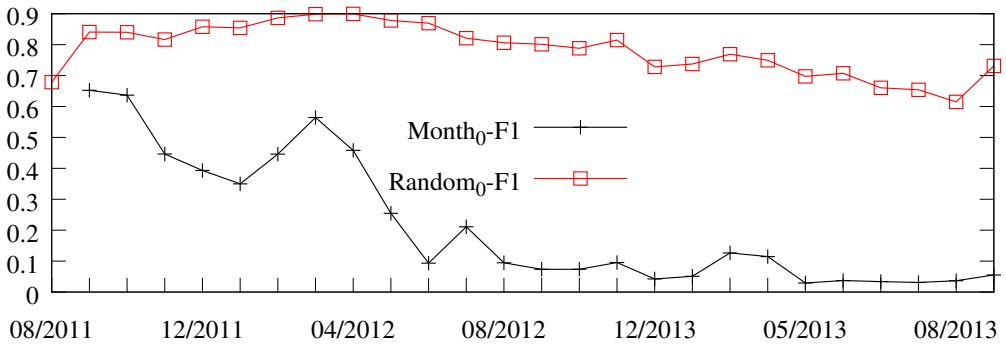
As described in Section 7.2.1, a key step of machine-learning approaches is the training of classifiers. The construction of the corresponding training dataset is consequently of importance, yet details about how it is achieved are largely missing from the literature, as was shown in Section 3.3.

There are two common selection patterns for training datasets: (1) use a collected and published dataset of malware, such as Genome, to which one adds a subset of confirmed goodware; (2) build the dataset by randomly picking a subset of goodware and malware from a dataset collected from either an online market or an open repository. Both patterns lead to the same situations: i.e. that *some items in the training dataset may be historically posterior to items in the tested dataset*. In other words, (1) the construction of the training set is equivalent to a random history-unaware selection from a mix of known malware and goodware; and (2) the history of creation/apparition of android applications is not considered as a parameter in assessment experiments, although the practice of malware detection will face this constraint.

Following industry practices, when a newly uploaded set of applications must be analyzed for malware identification, the training datasets that are used are, necessarily, historically anterior to the new set. This constraint is however eluded in the validation of malware detection techniques in the research literature. To clearly highlight the bias introduced by current assessment protocols, we have devised an experiment that compares the performance of the machine learning detectors in different scenarios. The malware detectors are based on classifiers that are built in two distinct settings: either with randomly-constructed training datasets using a process described in Figure 7.1 or with datasets that respect the history constraint. To reduce the bias

between these comparisons, we ensure that the datasets are of identical sizes and with the same class imbalance between goodware and malware. Thus to build a history-unaware dataset R_0 for comparing with training dataset constituted of data from month M_0, we randomly pick within the whole dataset the same numbers of goodware and malware as in M_0. We perform the experiments by training first on M_0 and testing on all following months, then by training on R_0 and testing on all months (cf. Figure 7.2).

Figure 7.3 illustrates the results of our experiments. When we randomly select the training dataset from the entire dataset and build classifiers for testing applications regrouped by month, the precision and recall values of the malware detector range between 0.5 and 0.85. The obtained F-Measure is also relatively high and roughly stable. This performance is in line with the performances of state-of-the-art approaches reported in the literature.



Reading: The $Month_0$ curve shows the F-Measure for a classfier trained on the month 0, while the $Random_0$ curve presents the F-Measure for a classifier built with a training set of same size and same goodware/malware ratio as month 0, but drawn randomly from the whole dataset.

Figure 7.3: Performance of malware detectors with history-aware and with history-unaware selection of training datasets

We then proceed to constrain the training dataset to be historically coherent to the test dataset. We select malware and benign apps in the set of apps from a given month, e.g., M_0, as the source of data for building the training dataset for the classification. The tests sets remain the same as in the previous experiments, i.e., the datasets of applications regrouped by month. We observe that as we move away from M_0 to select test data, the performance considerably drops.

We have repeated this experiment, alternatively selecting each different month from our time-line as the month from which we draw the training dataset. Using a training set that is not historically coherent always led to significantly higher performance than using a historically coherent training set.

Finding RQ-1: *Constructing a training dataset that is consistent with the history of apparition of applications yields performances that are significantly worst than what is obtained when simply randomly collecting applications in markets and repositories. Thus,* **without further assessment, state-of-the-art approaches cannot be said to be powerful in real-world settings**.

Finding RQ-2: *With random selections, we allow malware "from the future" to be part of the training sets. This however leads to biased results since the performance metrics are artificially improved.*

7.3.2 Lineages in Android Malware

Our second round of experiments has consisted in investigating the capabilities of a training dataset to help build classifiers that will remain performant over time. In this step of the study we aim at discovering how the variety of malware is distributed across time. To this end, we consider building training datasets with applications in each month and test the yielded classifiers with the data of each following months.

Figures 7.4 and 7.5 provide graphs of the evolution over time of, on the one hand, F-Measure and, on the other hand, Precision of malware detectors that have been built with a training dataset at month M_i and applied on months $M_{k,k>i}$. Disregarding outliers which lead to the numerous abrupt rise and breaks in the curves, the yielded classifiers have, on average, a stable and high Precision, with values around 0.8. This finding suggests that *whatever the combination of training and test dataset months, the built classifiers still allow to identify with good precision the malware whose features have been learnt during training*.

On the other hand, the F-measure performance degrades over time: for a given month M_i whose applications have been used for the training datasets, the obtained classifier is less and less performant in identifying malware in the following months $M_{k,k>i}$. This finding, correlated to the previous one, suggests that, over time, the features that are learnt in the training dataset correspond less and less to all malware when we are in the presence of **lineages** in the Android malware. We define a `lineage` as a set of malware that share the same traits, whether in terms of behavior or of coding attributes. Note that we differentiate the term `lineage` from the term `family` which, in the literature, concern a set of malware that exploit the same vulnerability. *Lineage* is a more general term.

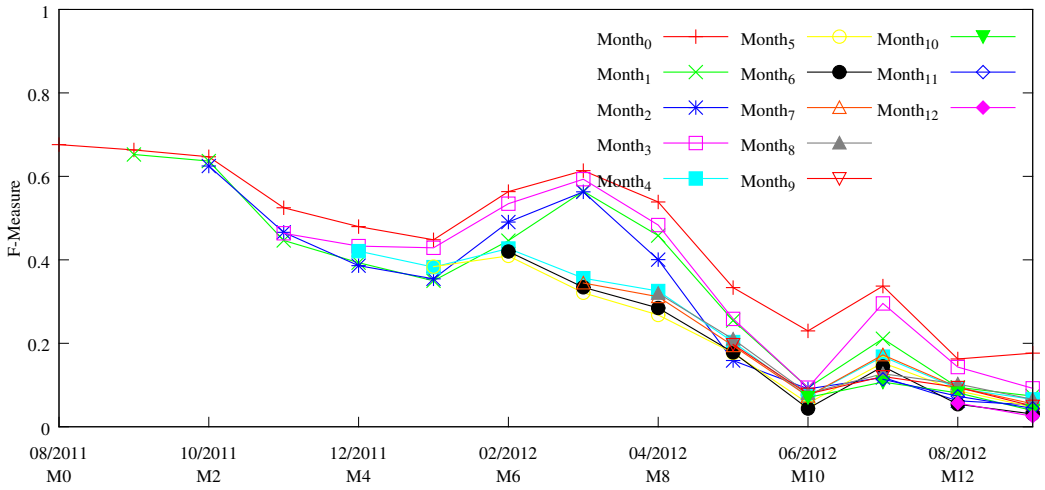


Figure 7.4: Performance Evolution of malware detectors over time

The experiments also highlight the bias introduced when training classifiers with a specific and un-renewed set of malware, such as the Genome dataset, which is widely used. It also confirms why the random selection of malware in the entire time-line as presented in Section 7.3.1, provides good performances: many lineages are indeed represented in such training datasets, including lineages that should have appeared for the first time in the test dataset.

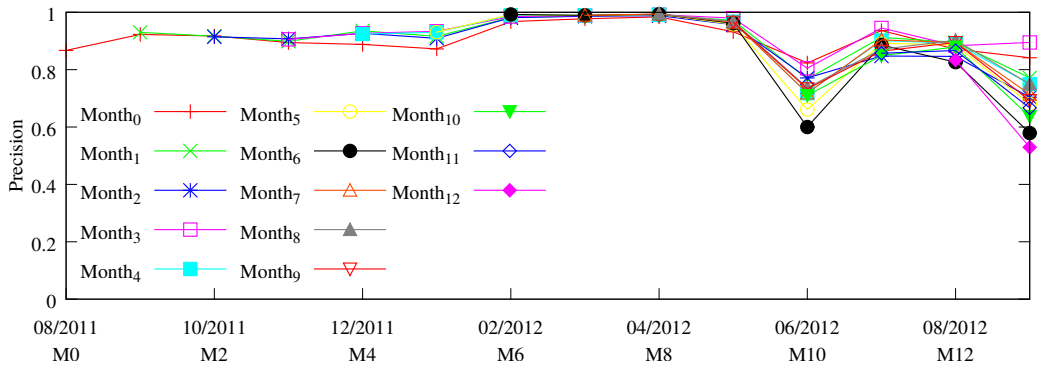


Figure 7.5: Evolution of Precision of malware detectors over time

Finding-RQ3: *Android malware is diversified. The existence of lineages complicates malware detection, since training datasets must be regularly updated to include a larger variety of malware lineages representatives.*

7.3.3 Is knowledge "from the future" the Grail?

Previous experiments have shown that using applications from the entire time-line, without any historical constraint, favorably impacts the performance of malware detectors. We have then proceeded to show that, when the training dataset is too old compared to the test dataset, this performance drops significantly. We now investigate whether training data that are strictly posterior to the test dataset could yield better performance than using data that are historically anterior (coherent). Such a biased construction of datasets is not fair when the objective is to actively keep malicious apps from reaching the public domain. However, such a construction can be justified by the assumption that the present might always contain representatives of malware lineages that have appeared in the past.

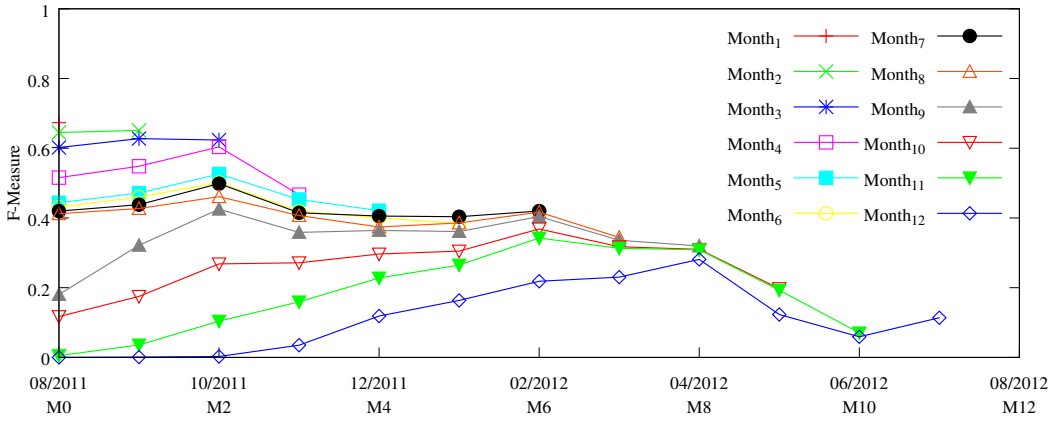


Figure 7.6: Performance of malware detectors when using recent data to test on old datasets

In the Android ecosystem, thousands of applications are created weekly by developers. Most of them, including malware from new lineages, cannot be thoroughly checked. Nevertheless, after some time, antivirus vendors may identify the new malware. Machine-learning processes can thus be used to automate a large-scale identification of malware in applications that have been made available for some time. Figure 7.6 depicts the F-Measure performance evolution of the malware detectors: for each month M_i, that is used for training, the obtained classifiers are used to predict malware in the previous months $M_{k,k<i}$. Overall, the performance is dropping significantly with the time difference between test and training datasets.

Finding-RQ4: *Apps, including malware, used for training in machine learning-based malware detection must be historically close to the target dataset that is tested. Older training datasets cannot account for all malware lineages, and newer datasets do not contain enough representatives of malware from the past.*

7.3.4 Naive Approaches to the Construction of Training Datasets

Given the findings of our study presented in previous sections, we investigate through extensive experiments the design of a potential research approach for malware detection which will be in line with the constraints of industry practices. At a given time t, one can only build classifiers using datasets that are anterior to t. Nevertheless, to improve our chances of maintaining performance, two protocols can be followed:

1. *Keep renewing the training dataset entirely to stay historically close to the target dataset of test. This renewal process must however be automated to remain realistic*: In this scenario, we assume that a bootstrap step is achieved with antivirus products at month M_0 to provide a first reliable training dataset. The malware detection system is then on its own for the following months. Thus, the classification that is obtained on month M_1, using month M_0 for training, will be used "as is" to train the classifiers for testing applications data of month M_2. This system is iterated until month M_n as depicted in Figure 7.7, meaning that, once it is bootstrapped, the detection system is automated and only relies on its test results to keep training new classifiers. In practice, such an approach makes sense due to the high precision values recorded in previous experiments.

2. *Include greedily the most knowledge one can collect on malware lineages*: This scenario is also automated and requires bootstrapping. However, instead of renewing the training dataset entirely each month, new classification results are added to the existing training dataset and used to build classifiers for the following month.

Figure 7.8 shows that the F-measure performance is slightly better for scenario 2. The detailed graphs show that, in the long run, the Recall in scenario 2 is indeed better while the Precision is lower than in scenario 1. In summary, these two scenarios exhibit different trade-offs between Precision and Recall in the long run: Scenario 1 manages to pinpoint a small number of malware with good precision while scenario 2 instead finds more malware at the cost of a higher false-positive rate.

While of little use in isolation, those scenarios provide insights through empirical evidence on how machine learning-based malware detection systems should consider the construction of training sets.

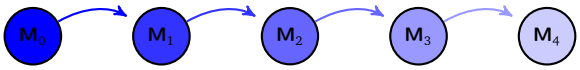


Figure 7.7: Using classification results of M_{n-1} as training dataset for testing M_n

Finding-RQ5: *Maintaining performance of malware detectors cannot be achieved by simply adding/renewing information in training datasets based on the output of previous runs. However, these scenarios have shown interesting impact on performance evolution over time, and must be further investigated to identify the right balance.*

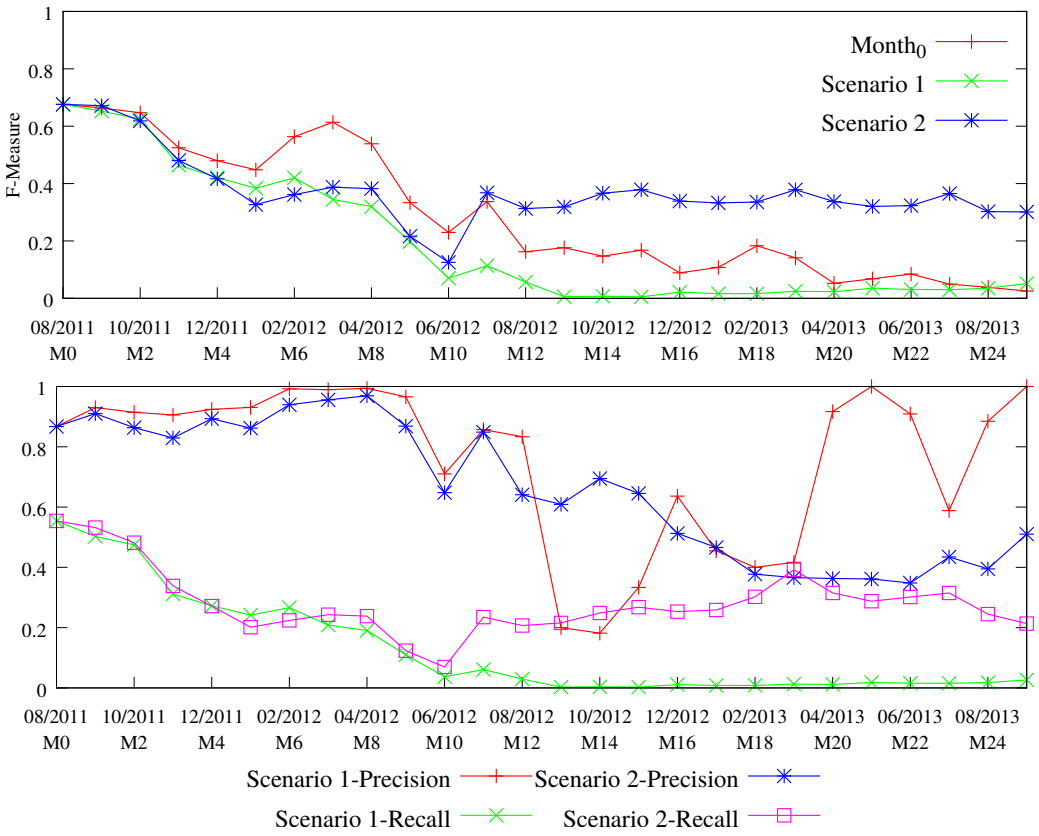


Figure 7.8: Comparing two naive approaches

7.4 Insights and Future work

7.4.1 Findings

(1) History constraints must not be eluded in the construction of training datasets of machine learning-based malware detectors. Indeed, they introduce significant bias in the interpretation of the performance of malware classifiers.

(2) There is a need for building a reliable, and continuously updated, benchmark for machine learning-based malware detection approaches. We make available, upon request, the version we

have built for this work. Our benchmark dataset contains about 200,000 Android applications spanning 2 years of historical data of Android malware.

7.4.2 Insights

(1) Machine-learning cannot assure the identification of an entirely new lineage of malware that is not represented in the training dataset. Thus, there is need to regularly seed the process with outside information, such as from antivirus vendors, of new lineages of malware.

(2) In real world settings, practitionners cannot be presented with a reliable dataset for training. Indeed, most malware are discovered, often manually, by antivirus vendors far later after they have been available to end-users Apvrille and Strazzere, 2012. Large-scale ML-based malware detection must therefore be used to automate the discovery of variants of malware which have been authenticated in a separate process.

7.4.3 Threat to Validity

To perform this study, we have considered a unique use-case scenario for using machine learning-based malware detection. This scenario consists in *Actively preventing malware from reaching markets* and is extremely relevant to most real-world constraints. Indeed, in practice, it is important to keep the window of opportunity very narrow. Thus, to limit the number of infected devices, Android malware must be detected as they arrive in the market. It is therefore important that state-of-the-art approaches be properly assessed, taking into account history constraints.

There is however a second use-case scenario which concerns online repositories for research and would consist on *cleaning such repositories regularly*. In this scenario, repositories maintainers attempt to filter malicious apps once a new kind of malware has been discovered. In such a context, practitionners can afford to wait for a long time before building relevant classifiers for identifying malware that have been since in the repository. Nevertheless, such repositories are generally of reasonable size and can be scanned manually and with the help of antivirus products.

There is a possibility that the results obtained in this chapter would not be reproduced with either a different feature set and/or a different dataset. Nonetheless, we have no reason to believe that the way the dataset was collected induced any bias.

7.4.4 Future work

(1) Building on the insights of our experiments, we plan to investigate how to maintain the performance of machine learning-based malware detectors until antivirus vendors can detect a new strain of malware. This research direction could help bring research work to be applied on real-world processes, in conjunction with antivirus products which are still widely used, although they do not scale to the current rates of malware production. (2) To account for the evolution of representations of malware lineages in training datasets over time, we plan to investigate a multi-classifier approach, each classifier being trained with more or less outdated data and weighted accordingly. A first challenge will be on how to infer or automate the choice of weights for different months in the timeline to build the most representative training dataset.

7.5 Conclusion

Given the steady increase in the adoption of smartphones worldwide, and the growth of application development for such devices, it is becoming important to protect users from the damages of malicious apps. Malware detection has thus been in recent years the subject of renewed interest, and researchers are investigating scalable techniques to spot and filter out apps with malicious traits among thousands of benign apps.

However, more than in other fields, research in computer security must yield techniques and approaches that are truly usable in real-world settings. To that end, assessment protocols of malware detection research approaches must reflect the practice and constraints observed by market maintainers and users. Through this empirical study we aim to prevent security research from producing approaches and techniques that are not in line with reality. Furthermore, given the performances reported in state-of-the-art literature of malware detection, while market maintainers still struggle to keep malware out of markets, it is important to clear the research field by questioning current assessment protocols.

In this chapter, we have investigated the relevance of history in the selection of assessment datasets. We have performed large-scale experiments to highlight the different bias that can be exhibited by different scenarios of dataset selection. Our main conclusion is that the assessment protocol used for current approaches in the state-of-the-art literature is far from the reality of a malware detection practice for keeping application markets clean. We have further investigated naive approaches to training dataset construction and drawn insights for future work by the research community.

Chapter 8

An investigation into the effect of Antivirus Disagreement

Contents

8.1 Why the lack of Antivirus Consensus is a problem

As noted in subsection 5.2.1, the different antivirus products often disagree on whether one application is malicious or not.

An important question arises: when should an application be called a malware? Intuitively, the safe is to assume an application is malicious as soon as at least one antivirus products flags it as a malware. However, antivirus products do commit errors, and sometimes report a file as malicious while it is not malicious. Symmetrically, applications whose maliciousness is established go undetected.

As a consequence, data coming from antivirus products must be considered as *noisy*. As an illustration, in our full dataset, 64.9% of APKs are not detected by any antivirus product. However, only 0.05% of APKs are detected as malicious by thirty or more antivirus products.

For a significant fraction of our dataset, there is no consensus among antivirus editors. This lack of clear consensus creates a twofold problem in the field of machine learning-based malware detection.

8.1.1 Training Material Quality

Using antivirus products as a source of labels for training data means providing a machine learning algorithm with imperfect data.

The algorithm will hence try to build a model of this data, and therefore is likely to *learn* the data imperfections, and to embeds those into the resulting model. Even a *perfect* model of the training set would not be able to perfectly classify a test set.

8.1.2 Testing Material Quality

The other side of this consensus problem lies with the evaluation of malware detectors. To evaluate the performance of a given malware detector, its predictions are compared to a *ground truth*.

Though, when the *ground truth* is known not to be an *absolute truth*, a malware detector whose predictions are fully in line with the *ground truth* cannot be qualified of *perfect* neither.

The effect of the lack of consensus among antivirus products is profound: How can we evaluate the performance of a malware detector if the only reference we can compare its predictions to is indeed imperfect?

Furthermore, not only is the reference imperfect, but it is imperfect to an extent that cannot be evaluated nor measured, thus preventing researchers from computing error bars associated with malware detector performance that are evaluated on the basis of such an imperfect reference.

8.2 Research Questions

Because we cannot obtain a perfect ground truth, it is impossible to directly quantify the effects of relying on a noisy reference.

We can however measure the impact of changing the level of consensus required to qualify one application as a malware.

For this study, we define a *Consensus Threshold* as being the number of antivirus products it takes to be qualified as a malware. For example, when using a threshold of 5, only apps that are detected as malicious by five or more antivirus products are considered malware.

Having a threshold superior to one raises another question. Keeping the example of a threshold of 5, is an application detected by four antivirus products a goodware? Or is it preferable to consider such an application as unknown? To take into account these two possible ways of handling the *grey area*, we also define the *Cleanness* of a reference classification: A reference is said *cleaned* when apps which have been detected as malicious by (1) at least one antivirus product but (2) less than *Threshold* antivirus products are removed.

Similarly, a reference is said *uncleaned* if the applications detected by less than *Threshold* antivirus products are considered to be goodware.

> **RQ 1**: Does removing from the training set apps that are detected by only a few antivirus help increase precision?

This first research question investigates whether apps detected as malware by only a small number of antivirus products constitute *noise* that pollutes the machine-learning process. If such is the case, we expect that removing, or reducing, this noise would lead to higher precision.

> **RQ 2**: What is the impact on performance metrics when using for training a different consensus threshold than for testing?

8.3 Code Metrics-based Features Set

We developed a Feature Set using heuristics and elements of coding style.

8.3.1 Heuristics Features

Several characteristics are likely to carry discriminating power for detecting malware. For example, it has been shown (Liu & Liu, 2014) that malware are more likely than goodware to request specific permissions. We therefore include the list of requested permissions in this feature set.

Moreover, we include whether or not an application *uses* a few specific, manually selected Android API features. Amongst such features are:

Usage of dynamic code Android allows apps to load code at run time. Android apps can load DEX bytecode, or native Linux `.so` libraries. Since the loaded code can either be shipped inside the APK or be downloaded at run time from Internet, it can be used by malware authors as a tool to hinder analysis of the application. Indeed, malicious code is very hard to detect when it is not inside the APK but downloaded on the fly, hence evading static analysis.

Reflection Code Often used together with dynamic code loading, usage of the Java Reflection API can also be used to try to hide information from security analyst.

Cryptographic API usage Some malware try to hide their code by using packers or by encrypting code that will be dynamically loaded once unencrypted. We therefore record the use of the `javax.crypto` and `java.security.spec` packages.

ASCII Obfuscation

SMS API SMS has been used in several malware, and is a very simple way for an attacker to gain money. The usage of the two classes `android.telephony.SmsManager` and `android.telephony.SmsMessage` are monitored.

Telephony API Similarly, we detect whether an app uses the facilities provided by the Android Framework to emit and receive phone calls.

Account Manipulations An Android system may store credentials for various services such as Google, Facebook or Twitter. We check whether an application makes use of this facility.

Camera By detecting whether the Android API for managing the camera is called, we know if an application is able to take pictures.

NFC Many recent Android devices now embed an NFC (Near Field communication). Since NFC can be used to transfer data from or to a device, the usage of this function is also included in our feature set.

Location Acquisition The `android.location.LocationManager` class allows an application to know the location of the device and as such its detection is also part of our feature set.

Low level Networking The creation of client or server TCP or UDP sockets is detected.

8.3.2 Coding Style Features

The other family of characteristics added to our feature set is built from statistics on various aspects of the application's bytecode. Included in this family of features are:

Size of Dex The total size in bytes of the bytecode.

Number of permission uses Different from the number or permissions *requested*, this feature records the number of times Android framework methods that need a permission are called.

Number of Classes The total number of classes declared by the application.

Number of methods The total number of methods declared by the application.

Number of Activities / Services / Providers These three specific types of Android application components are each counted.

Number of Strings and Number of Fields We record the number of declarations of Strings and of Fields.

Also used in this feature set are statistics on the length of strings and methods:

Strings Length Using all the Strings defined by the application, we compute the mean string length, the standard deviation, the median and the maximum value.

Methods Length Similarly, we compute the mean, standard deviation, median and maximum value of the length of methods. The length of a method is the number of bytes of its bytecode definition.

For named objects, we also add to this feature set statistics on objects' name length:

Class Name Length Mean value, standard deviation, median and maximum value for Class name length.

Field Name Length Mean value, standard deviation, median and maximum value for Field name length.

Method Name Length Mean value, standard deviation, median and maximum value for Method name length.

Single Character names A strong sign of code obfuscation, we compute the fraction of Class names, and the fraction of Method names that are only one character long.

Finally, we include *normalised* values for permission usage:

Permission per Classs (resp. per Method) The number of permissions *requested* divided by the number of classes (resp. Methods[1]).

Permission Usage per Class (resp. per Method) The number of permissions *usages* divided by the number of Classes (resp. Methods[1]).

8.3.3 Feature Set Summary

All in all, this feature set uses 179 features, of which 128 are binary. This is a very small number of features when compared to the basic block-based feature set used in chapter 6 and chapter 7.

This small number of features has many advantages. First, the feature matrix files generated are orders of magnitude smaller than matrices representing our Basic Block-based feature set or any other feature set similar to n-grams. As a consequence, The I/O load is much more manageable, especially since it removes the need to perform feature evaluation and selection, whose only purpose is to reduce the number of features.

Furthermore, the small number of features means that both the training phase and the classification phase use less memory. Indeed, with our basic block-based feature set, we sometimes needed several hundreds Gigabytes of RAM for the training phase, rendering the whole process highly unpractical and setting an undesirable limit to the maximum size of the training dataset.

Finally, smaller feature matrices usually imply vastly reduced training time. Those aspects of low storage needs, low I/O needs, low memory needs and fast training all contribute to make this feature set practical to use in large-scale Malware detection schemes.

[1] For Permissions requested per Method and permissions usages per Method, the values are multiplied by 100 in order to obtain values significantly different from zero, which is necessary for Weka because its use of low precision real numbers would lead nearly all values to be rounded to zero

8.4 Experimental Setup

All our experiments presented here were conducted with the Random Forest algorithm as implemented in the `randomForest` package[2] of the R software[3].

8.4.1 Dataset

For the experiments described in this chapter we use as a base for training set all applications from our collection (see chapter 4) compiled during the month of June 2014. Apps compiled the following month—July 2014—constitute our test set.

For both cases, we had to discard apps for which we could not obtain a detection report from VirusTotal and apps that failed to pass through our feature extraction code. As illustrated in Table 8.1, the applications excluded represent only a very small fraction of our dataset.

Eventually, our training set is made of 179 028 unique APKs, and our test set contains 232 386 APKs.

Table 8.1: Dataset overview

	Total apps in this month	Missing VT report	Unable to extract features	Final Dataset
Train Set	186 816	6201	1587	**179 028**
Test Set	242 327	8775	1166	**232 386**

8.4.2 Parameters

To investigate the effects of antivirus disagreement, we set up a large-scale collection of Machine-Learning experiments where four parameters are tuned:

src_threshold This parameter is the number of antivirus products it takes to declare one application a malware in the **training set**. Only applications detected as malware by at least

[2] https://cran.r-project.org/web/packages/randomForest/index.html
[3] https://www.r-project.org/

`src_threshold` antivirus products are labelled with the class MALWARE in the training matrix.

`tgt_threshold` Similarly, this parameter controls how many antivirus are required to call one application a malware, but when evaluating the results of the classification of the **test set**. One application predicted as a malware will account as a True Positive only if it is detected as a malware by at least `tgt_threshold` antivirus products.

`src_cleaned` This boolean parameters controls the *cleanness* of the training set. When set to `True`, applications from the **training set** that are detected by at least one antivirus product, but fewer than `src_threshold` antivirus products are **removed**, as discussed in section 8.2. When set to `False`, all applications detected by fewer than `src_threshold` antivirus products are instead labelled as GOODWARE.

`tgt_cleaned` Symmetrically, this boolean parameters controls the *cleanness* of the test set. When `tgt_cleaned` is set to `True`, every application detected as malware by more than one but less than `tgt_threshold` antivirus products will be removed from the test set, and hence will not be classified.

While the maximum number of antivirus products used by VirusTotal can be as high as 57 [4], we only varied the `src_threshold` and `tgt_threshold` parameters from 1 to 30. Indeed, as previously noted in chapter 5, it is very rare to have 30 antivirus products agreeing on one sample being a malware. For example, in our training set, only two apps are flagged as malware by 30 or more antivirus products, which is clearly not enough training material to build a malware detector.

Overall, we derived 59 variants of training set (30 values of `src_threshold` multiplied by 2 values of `src_cleaned`, minus one to take into account that when `src_threshold` $= 1$, `src_cleaned` has no impact) and thus trained 59 models, each of which was tested against each of the 59 variants of test sets we similarly derived.

Our classifiers were each made so that their output for each sample from the test set is a real number and not only a malware/goodware binary class. It is expected that this real number—ranging from zero to one—would act as an indicator of the likeliness of being a malware.

A consequence of this choice is that there are more than just one set of performance metrics for a classification experiment. There is instead one set of performance metrics per every possible value of the real number—called the Cutoff.

Hence several curves shown in this chapter are plotted against the cutoff, which allows to know the range of possible values for any given performance metric, and to see its evolution with regards to the cutoff value.

[4] The number of antivirus products changes over times as VirusTotal integrates more and more products

8.5 Empirical Results

8.5.1 Research Question 1

To investigate whether cleaning training sets results in higher precision, we compare the precision values yielded by classifiers trained on several variants of our base training set, and all tested against our full, uncleaned reference test set (`tgt_threshold` = 1).

As is visible in Figure 8.1, cleaning training sets does indeed increase the precision. Moreover, it appears that the precision increases as more and more applications are filtered out (that is to say when `src_threshold` is increased).

Figure 8.2 zooms in the very top of Figure 8.1. We can see that the higher precision coming with increased *cleanness* is verified over the entire range of cutoff values, with the exception of the classifier built on `src_threshold` = 25 whose performance is more erratic, as expected given that it is trained on only two malware samples.

Cleaning a dataset essentially means removing from the training set samples whose malware classification is deemed untrustworthy. Therefore, this first result seems to indicate that those apps constituting the grey area are indeed significantly *polluting* the training phase.

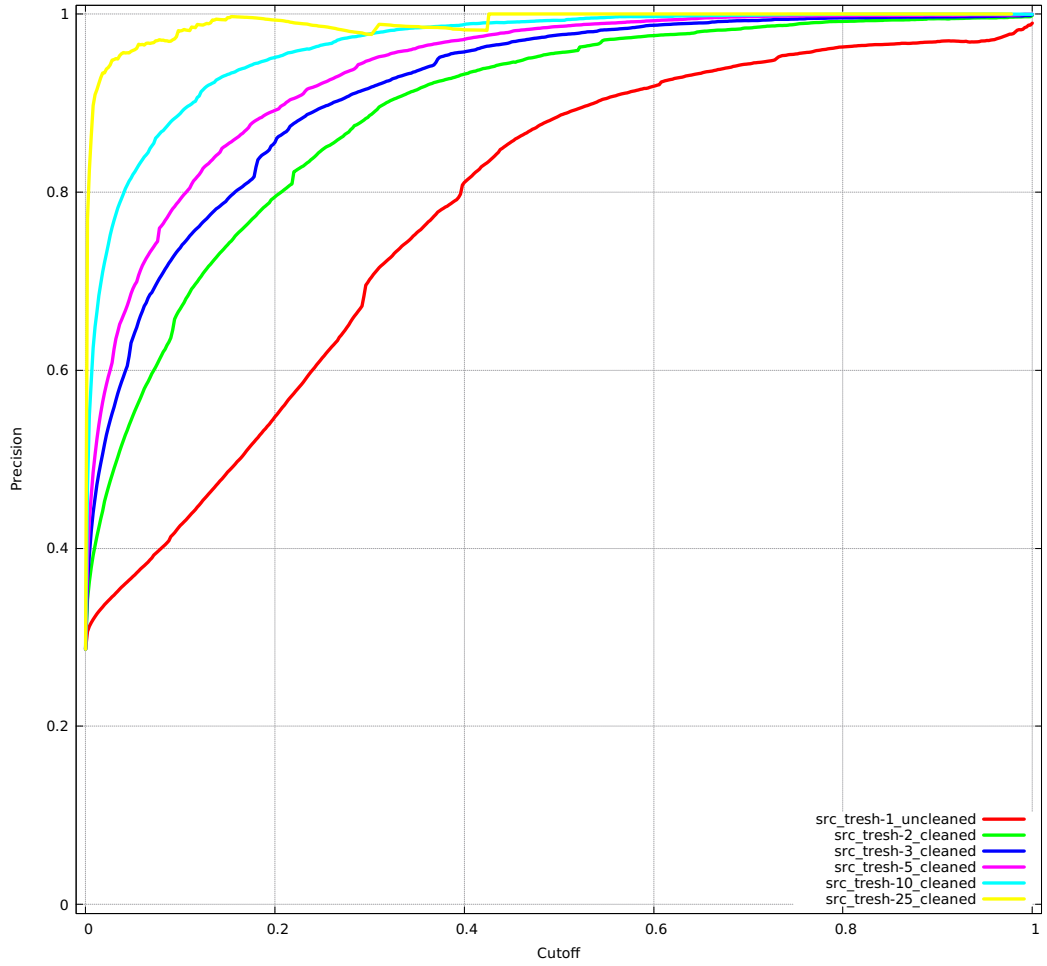


Figure 8.1: Comparison of Precision values for **cleaned** training sets with different threshold, all tested against a test set built with `tgt_threshold` = 1.

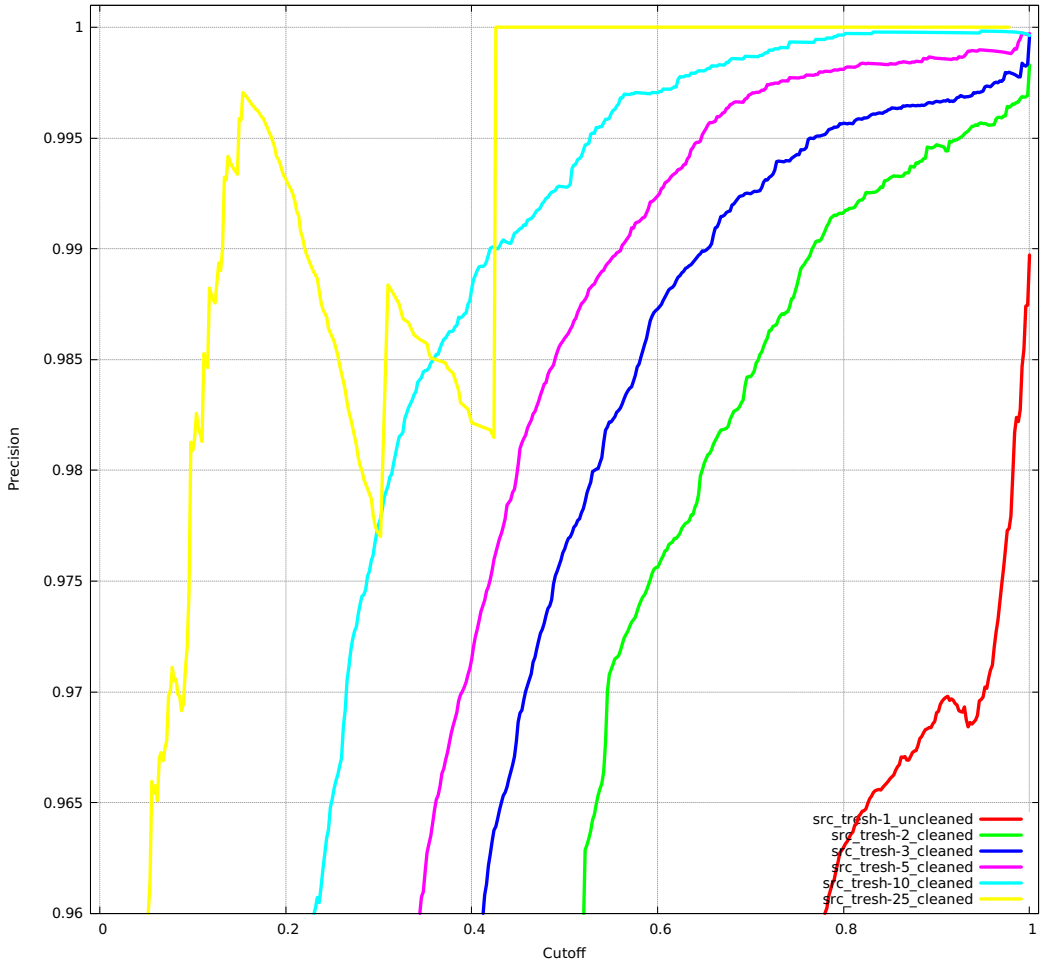


Figure 8.2: Zoom – Comparison of Precision values for **cleaned** training sets with different threshold, all tested against a test set built with `tgt_threshold` = 1.

Figure 8.1 and Figure 8.2 show that the precision increases with the `src_threshold` parameter. However, this behaviour could be independent of the cleanness. To test this hypothesis, we compare the precision of both cleaned and uncleaned testing sets for several given `src_threshold`.

Figure 8.3 shows that unless the threshold used is very high, the cleaned version of a training set yields a lower precision (dashed lines) than its uncleaned counterpart (plain lines).

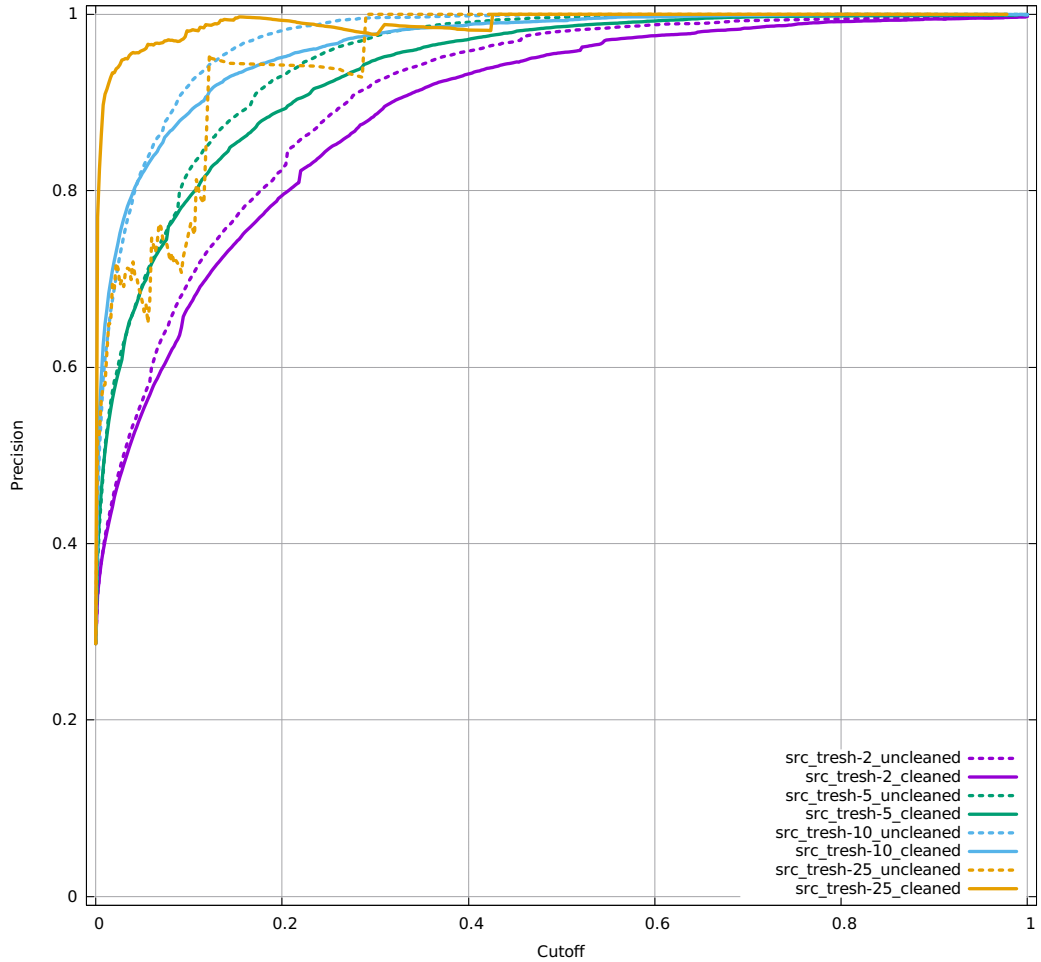


Figure 8.3: Zoom – Comparison of Precision values for **cleaned** training sets with different threshold, all tested against a test set built with `tgt_threshold` = 1.

Finding 1: For a given `src_threshold`, cleaning the training set usually leads to lower precision.

Also, Figure 8.3 allows us to confirm our earlier observation:

Finding 2: The precision usually increases with `src_threshold`.

8.5.2 Research Question 2: Impact of different consensus thresholds

It is important to understand that increasing `tgt_threshold`, all other things being equal, should have a tendency to result in lower precision, since many apps that used to be considered malware would then be considered goodware (if `tgt_cleaned` $= False$), or would not appear in the test set anymore (if `tgt_cleaned` $= True$). In both cases, increasing `tgt_threshold` reduces the number of malware in the test set, thus also reducing the likelihood of a random application taken from the test set to be a malware.

Figure 8.4 presents as heat-maps the Area Under the ROC Curve (AUC) for all the 3600 combinations of `src_threshold`, `tgt_threshold`, `src_cleaned` and `tgt_cleaned`, grouped in four quadrants, one for each of the four possible (`src_cleaned`, `tgt_cleaned`) couples.

It can be noticed from Figure 8.4 that the cleanness of the training set and/or of the test set has little impact on the overall shape of the AUC heat-map, as all four quadrants exhibit the same trends. However, the overall performance of classifiers decreases slightly faster with increases of `src_threshold` when `src_cleaned` $= False$, (i.e., on the two bottom heat-maps) than when `src_cleaned` $= True$.

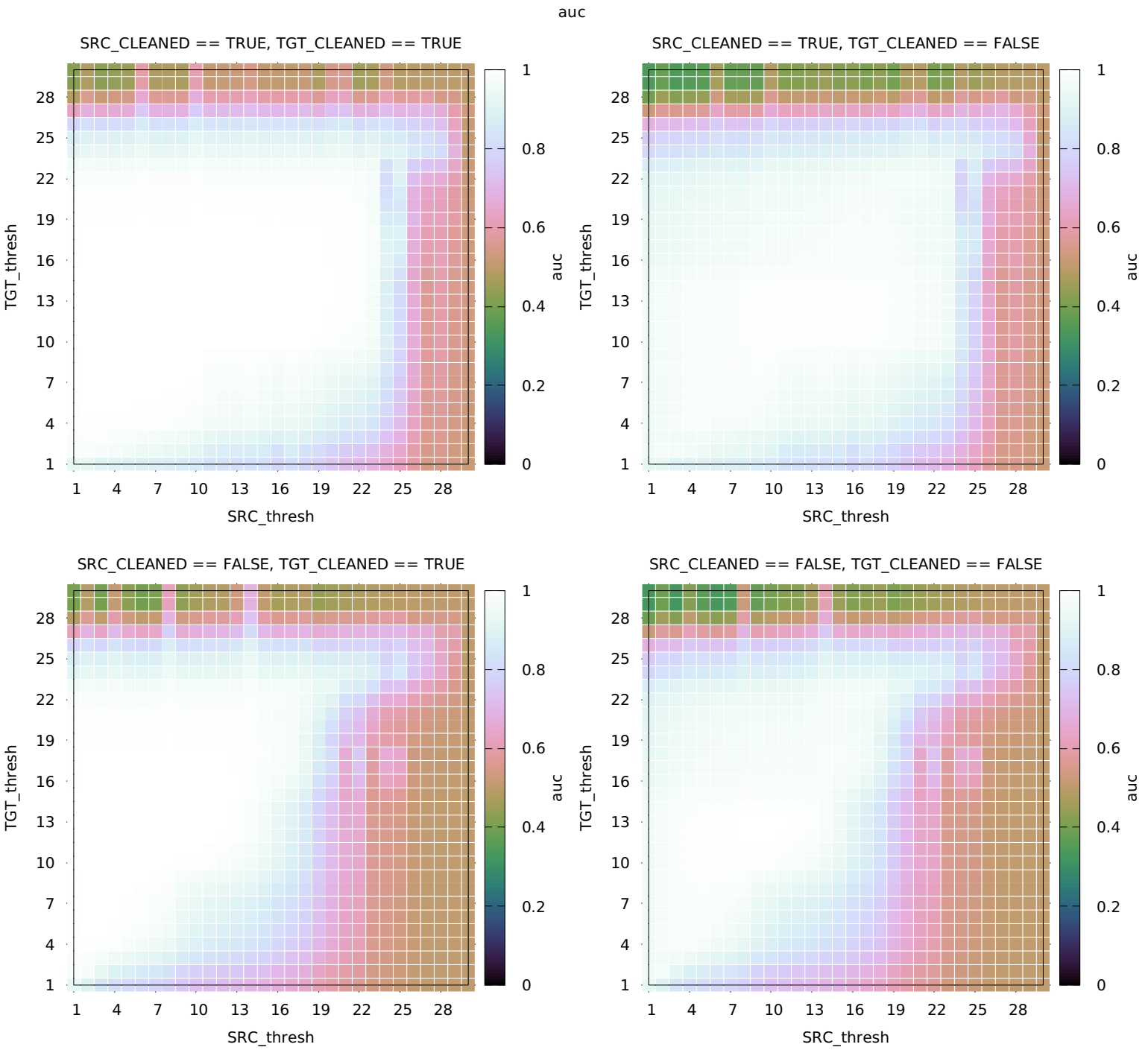


Figure 8.4: Comparison of AUC values

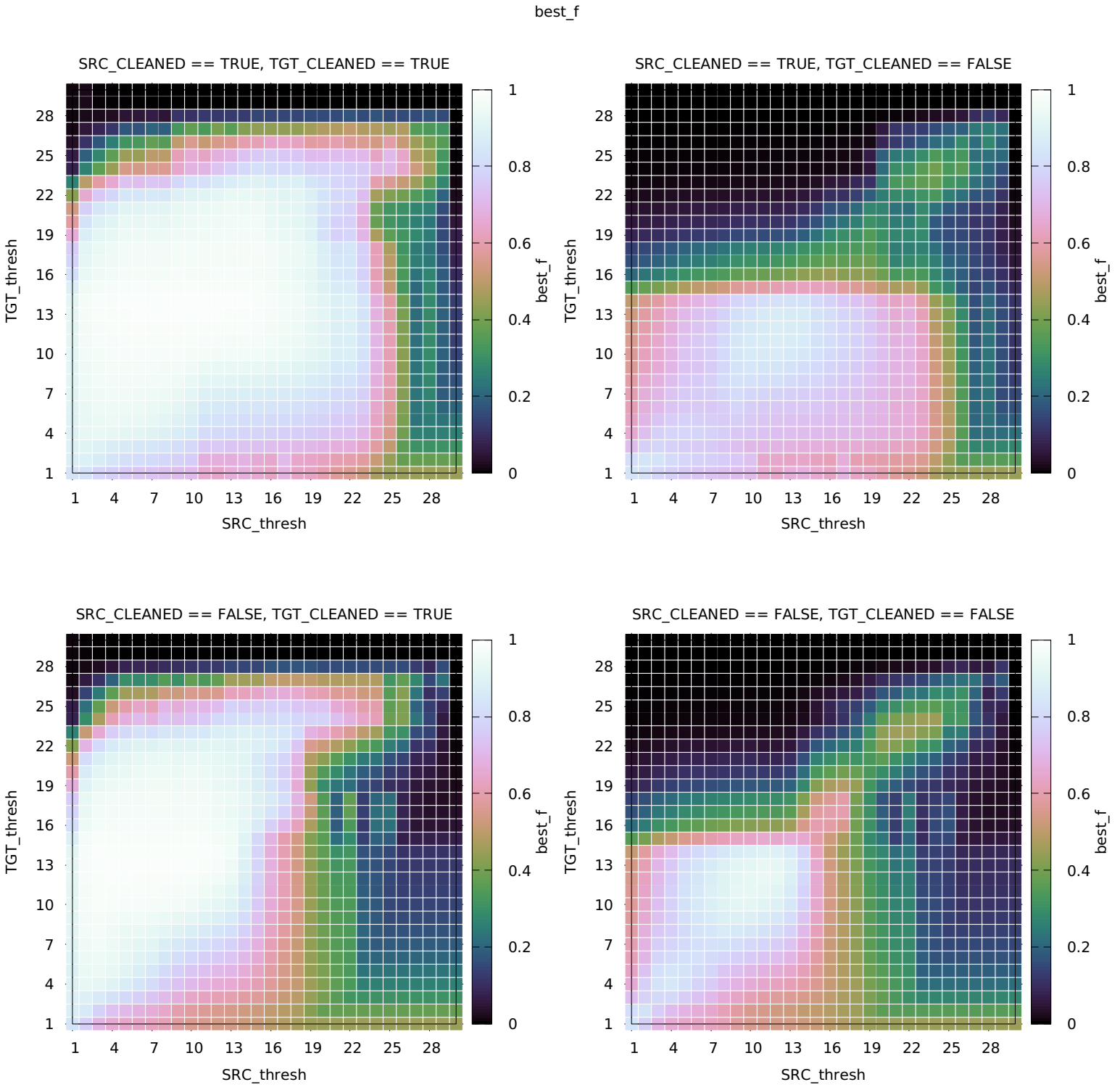


Figure 8.5: Comparison of F-measure values

Figure 8.5, which plots the best F-Measures obtained, allows us to continue our analysis, and shows that there is a trend around the diagonal where `src_threshold` = `tgt_threshold`. Indeed, it seems that the closer `src_threshold` is to `tgt_threshold`, the higher the F-Measure.

It also confirms a trend that was only lightly visible on the previous figure: The best results are obtained in the bottom left part of heat-maps, that is to say when `src_threshold` and `tgt_threshold` are both small.

Another confirmation is that even when `src_threshold` = `tgt_threshold`, not all values of threshold lead to similar performance metrics. Our approach is better at modelling datasets with smaller thresholds. We see two possible reasons that could explain this fact:

- When the threshold is higher, there are fewer malware to learn from—potentially not enough to learn a useful model;
- Higher thresholds *cannot* be modelled because they do not match any real differences with our feature set. This would be the case if, the set of applications detected by, for example, 25 antivirus products is random – so much random that the set of such malware in the training set share no characteristics, at least within our feature set, with the set of such malware in the test set.

While the later possibility is hard to conclusively dismiss, it seems unlikely. Furthermore, the first possibility is well in line with the issue of training set size we established in chapter 6.

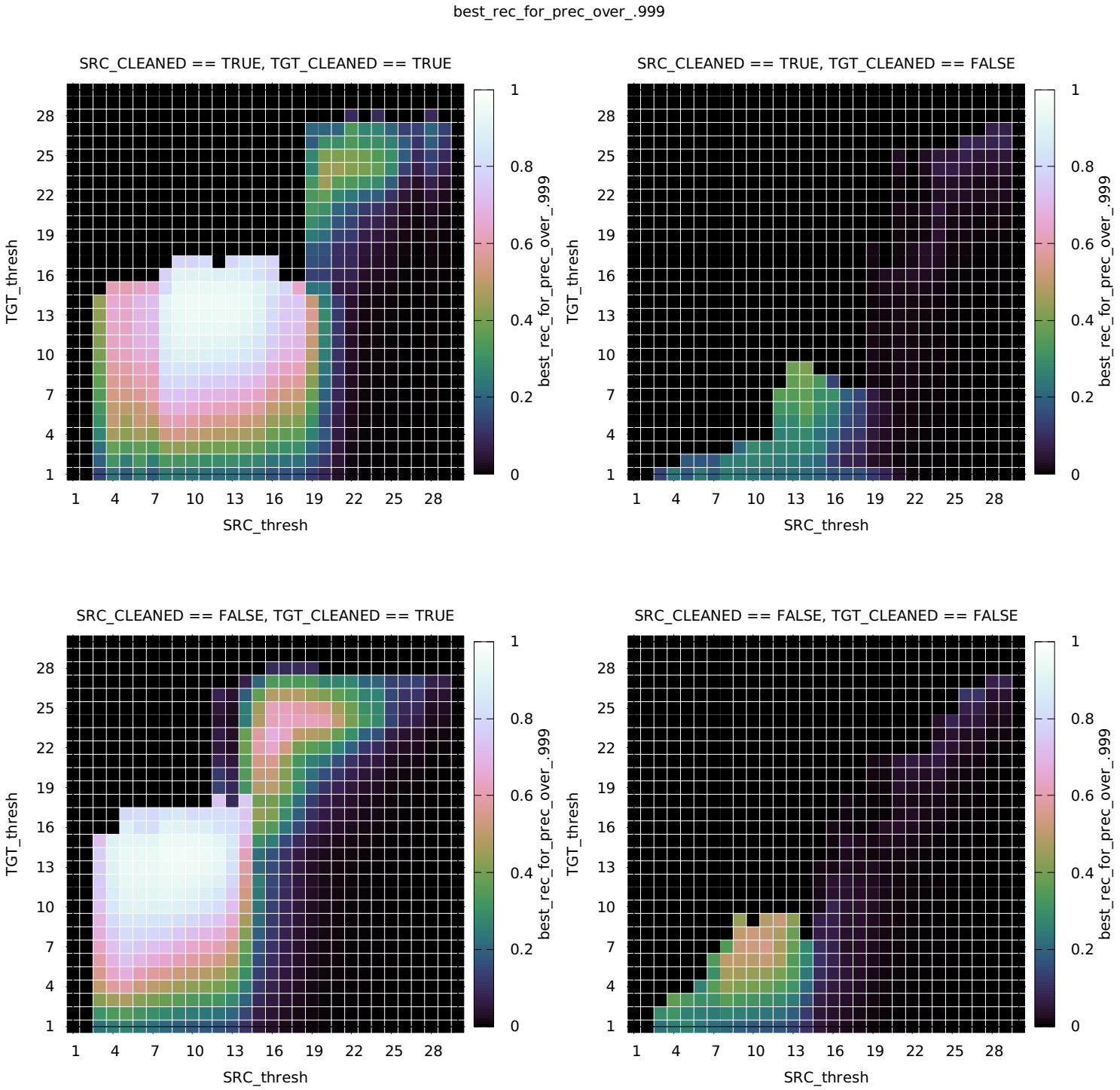


Figure 8.6: Comparison of the best Precision when $Recall > 0.999$

Figure 8.6 displays the Recall values when the Precision is constrained above 0.999.

We notice, by comparing the two heat-maps on the right-most column, that this performance metric is consistently higher when the training set is not cleaned, at least when the reference is not cleaned.

The two left-most heat maps are also interesting: they show that a very high Precision and a very high Recall can be obtained for the applications detected by around 10 to 16 antivirus editors by training a model with a `src_threshold` between 4 and 13.

8.6 Conclusion

The Consensus threshold chosen when performing a malware detection experiment has an important impact on the performance results that will be obtained.

Of course, there are no *good* consensus thresholds nor there are *bad* ones. Here, we simply demonstrated that one given approach can perform very differently when used with different consensus thresholds.

In the process, we also discovered that those differences in performance can be put to good use. Indeed, our results showed that training a malware detector with a threshold comprised between four and thirteen leads to a classifier that exhibits a very high precision on a subset of malware.

Hence this notion of consensus threshold is important, and not documenting how one approach reacts to changes of the threshold should be considered by experimenters as a limitation in the evaluation methodology.

Part IV

The Future of Machine Learning-Based Malware Detection

Chapter 9

A Time-Based Multi-Classifiers Approach

Contents

9.1 Introduction

The approach proposed in this chapter tries to work around several common issues in malware detection experiments.

As noted in chapter 7, History is an important aspect of a malware detection experiment. While we showed that recent data yields better results than old data, it does not mean by any way that old data should not be used as training material.

However, we also showed that simply having a giant training set containing as many apps as possible may not be a solution for leveraging recent and old data combined. In addition to performance issues, this would also be prohibitively expensive in terms of computational resources. Indeed, the time needed to train most Machine-Learning algorithms usually increases faster than the number of samples in the training set. Furthermore, it is frequent when performing machine learning-based malware detection experiments on large datasets for the training phase to require several hundreds gigabytes of memory, thus either limiting the scale of possible experiments or requiring expensive servers.

Practitioners hence have an incentive to try to limit the size of training sets, and at the same time it is desirable to use as much knowledge as possible.

In this chapter, we propose to combine several classifiers, each trained on one time span.

9.2 Experiment Design

For all the experiments performed in this chapter, the feature set used is the Code Metrics-based feature-set described in section 8.3.

Our experiments were carried over a set of 976 741 Android applications collected from the official Android market and from alternative markets. This set contains apps that were compiled during the first seven months of 2014. Besides ignoring 32 083 apps for which we could not obtain a detection report from VirusTotal[1] and 6 691 apps for which our tool failed to extract features, no selection nor sampling was made: Those 976 741 apps are all the applications from the first seven months of 2014 that a) we had access to and b) we could obtain a detection report for and c) we managed to process.

[1] VirusTotal enforces a 32MB file size limit

This dataset was grouped by months according to applications' compilation date, and we note M_1 to M_7 each of the seven months. By extension, M_x is also used throughout this chapter to denote either the set of applications compiled in month x or the feature matrix representing the applications compiled during month x. For each of these seven months, the number of applications is detailed in Table 9.1.

Table 9.1: Dataset overview

	Number of Apps
M_1	**114 644**
M_2	**89 768**
M_3	**109 911**
M_4	**116 275**
M_5	**134 736**
M_6	**179 028**
M_7	**232 386**

Month 1 to 6 will each be used as a training set to build one classifier. For the remainder of this chapter, we call *mono-classifier* these six classifiers each trained on one single month worth of data. We then try different combinations of these mono-classifiers and compare their performance when tested against Month 7.

In this chapter, all our classifiers—mono-classifiers and combinations of mono-classifiers alike—attribute a real number to each application of the test set, instead of just providing a goodware/malware flag. This real number outcome can take values from 0 (meaning that according to the algorithm, this application is very unlikely to be a malware) to 1 (in which case the algorithm considers this application very likely to be a malware).

9.3 Empirical Results

9.3.1 Baseline

To test whether or not our multi-classifiers provide a performance improvement, we first build a mono-classifier trained only on one month (M_6), and we apply this classifier to our test set (month M_7). The performance of this classifier will be used as a baseline to which we will compare the performance of our combinations of classifiers.

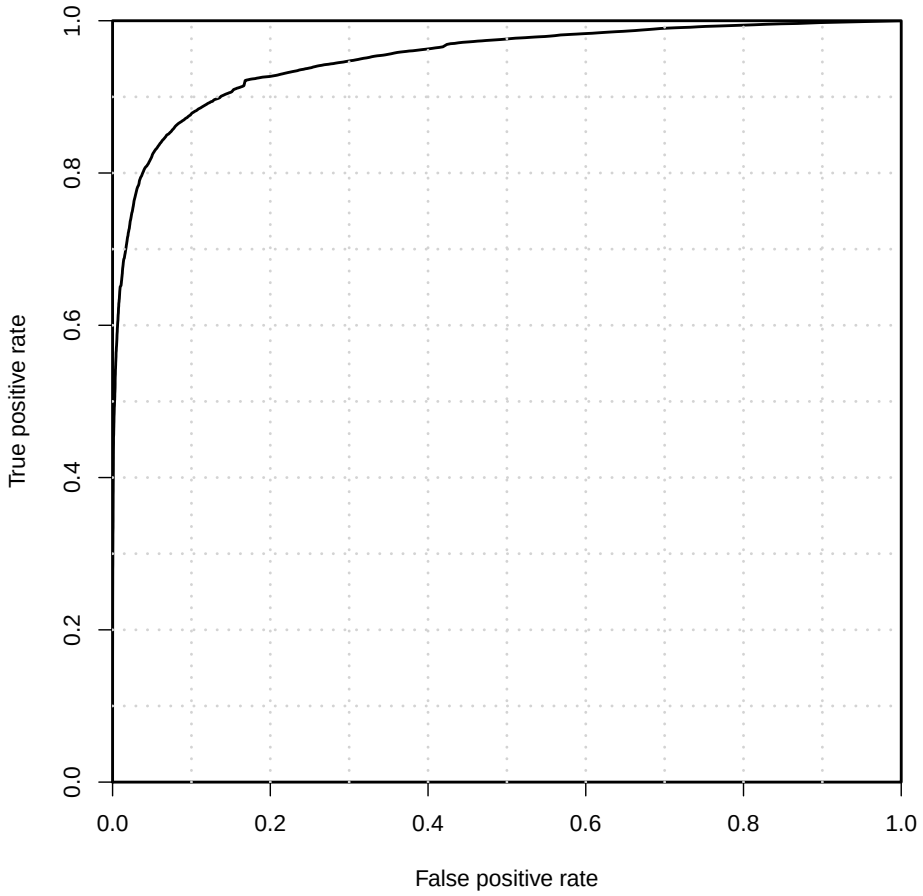


Figure 9.1: ROC Curve for the baseline classifier

Figure 9.1 presents the ROC curve obtained with the baseline classifier.

With an F-measure up to 0.85 and an AUC (Area Under the ROC Curve) over 0.95, this classifier, denoted in the rest of this section C6 (classifier trained on month M_6) demonstrates that our feature set has malware discrimination power.

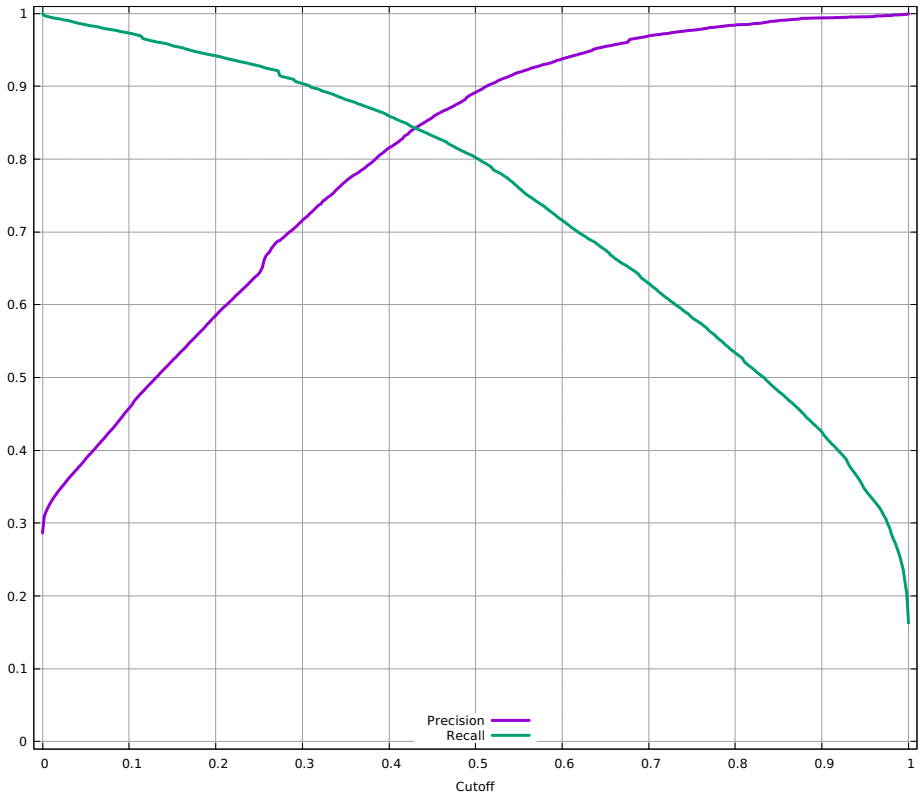


Figure 9.2: Precision and Recall against Cutoff for the baseline classifier

The evolution of Precision and Recall with regards to the Cutoff is visible on Figure 9.2. The best F-measure is obtained for a Cutoff value of 0.46, and half the malware can be detected with a precision of 0.988 at Cutoff=0.83.

9.3.2 Classifiers Combinations

Figure 9.3 shows ROC curves when combining up to 6 classifiers, each trained on a different month. Here, we build our multi-classifiers by averaging the outcome values of all underlying classifiers.

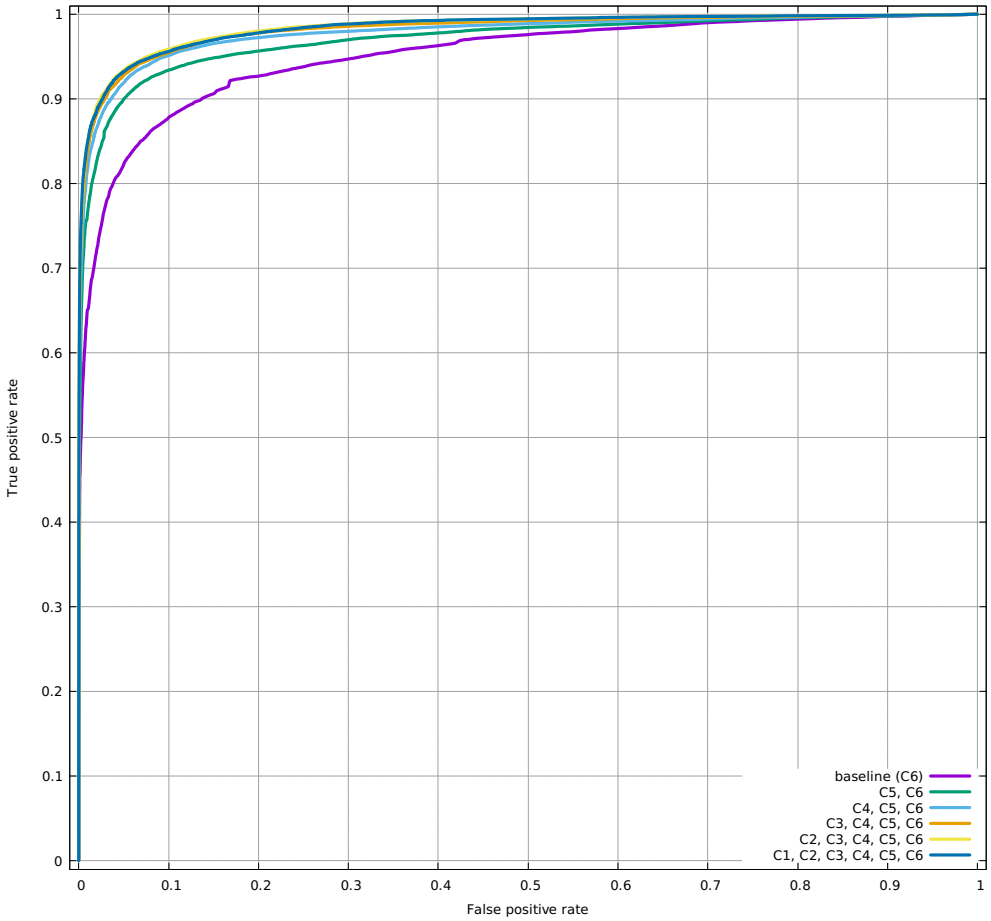


Figure 9.3: ROC Curve comparison

The general trend appearing from Figure 9.3 seems to confirm our insight: Adding more and more classifier results in better ROC curves, indicating better performance.

However, as is more visible in Figure 9.4 that zooms in the top-left part of Figure 9.3, our multi-classifiers that combines six classifiers does not follow this trend. Indeed, the ROC curve for the multi-classifier combining six classifiers wanders below the ROC curve for both the multi-classifier built on five classifiers and the one built on four.

The AUC values for each multi-classifier is displayed in Table 9.2. It shows that the AUC difference between our multi-classifiers built on 5 and 6 mono-classifiers is very small (around 4.10^{-4}).

The data we have is not sufficient to establish neither of the two following hypotheses:

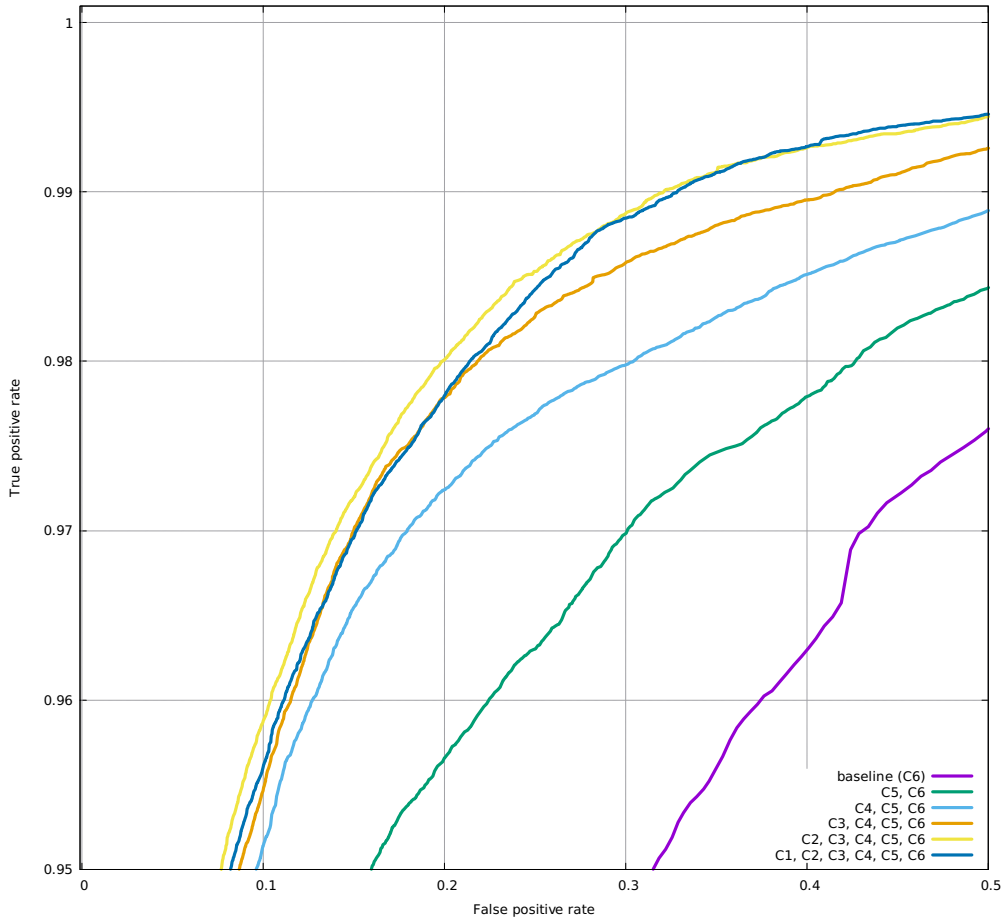


Figure 9.4: ROC Curve comparison – ZOOM

- The performance metrics improve when increasing the numbers of classifiers combined, but with diminishing returns.
- The performance metrics improve when increasing the numbers of classifiers combined, but only up to a unknown number of classifiers.

Table 9.2: Performance Metrics Comparison

Classifier	AUC	Best F-Measure (@Cutoff)	Best Precision for $Recall > 0.5$	Best Precision for $Recall > 0.99$	Best Recall for $Precision > 0.99$
Baseline	0,951	0.845 (@0.462)	0.988	0.361	0.481
Combination of 2 classifiers	0.970	0.893 (@0.438)	0.995	0.381	0.631
Combination of 3 classifiers	0.978	0.907 (@0.427)	0.997	0.425	0.701
Combination of 4 classifiers	0.981	0.914 (@0.44)	0.998	0.489	0.746
Combination of 5 classifiers	0.9841	0.919 (@0.424)	0.998	0.553	0.758
Combination of 6 classifiers	0.9837	0.916 (@0.411)	0.999	0.547	0.769

Table 9.2 lists several performance metrics and shows that the combination of six classifiers can manage to recall 77% of malware with a precision over 0.99—outperforming all other combinations for that specific performance metric.

9.4 Combining Classifiers

In the previous section, the multi-classifiers were built by computing an unweighted average of mono-classifiers.

In this section, we compare the performance obtained by the combination of six classifiers when the method of combining the results of mono-classifiers is changed.

We explore several methods[2] of combining classifiers:

mean An unweighted mean of mono-classifiers (as performed in the previous section);

min The final outcome is the lowest value amongst the mono-classifiers;

max The final outcome is the highest value amongst the mono-classifiers;

[2] We do not consider the sum of all mono-classifiers' values since for our purpose here, it is strictly equivalent to the unweighted mean i.e., both rank apps in the exact same order

Product The final outcome is the product of all mono-classifiers' values.

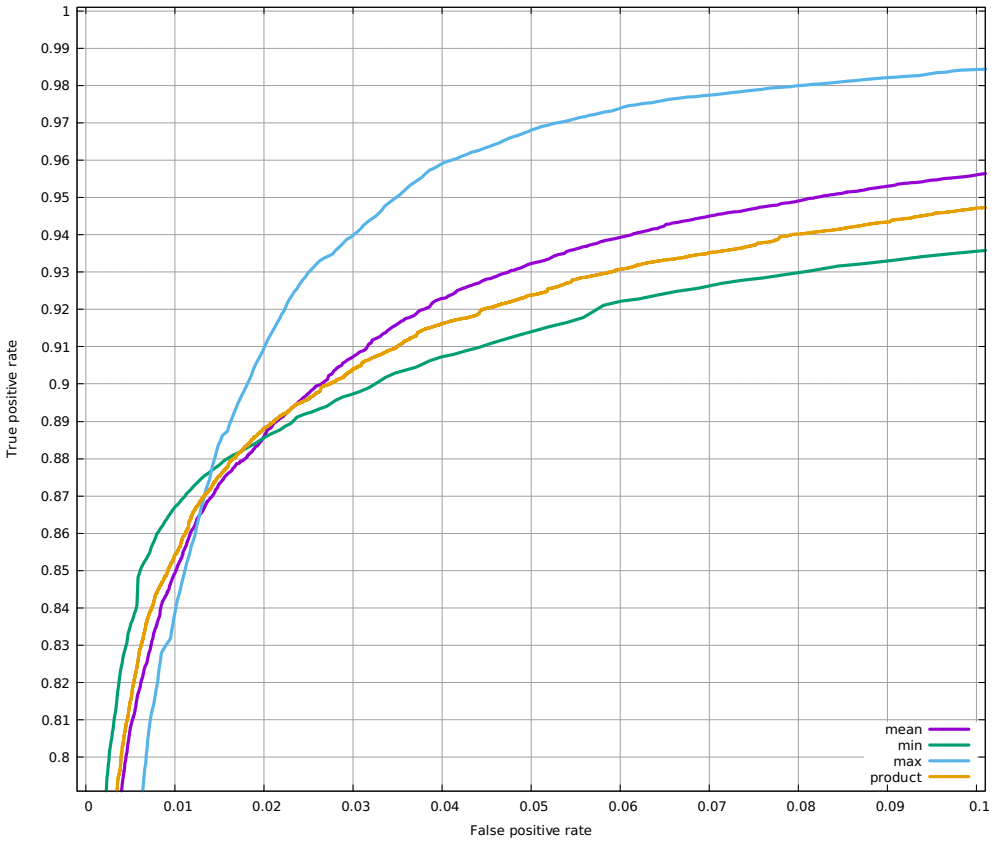


Figure 9.5: ROC Curve comparison for the different combining methods – ZOOM

Figure 9.5 presents the ROC curves for four different classifiers, each built from the six same mono-classifiers, but combined in different ways.

Table 9.3 shows various performance metrics for different combining methods. We notice that for different combining methods, the best F-Measure is obtained for Cutoff values that are very different, thus indicating that the evolution of Precision and Recall against Cutoff is significantly different.

Table 9.3: Performance Metrics Comparison for different Combining Methods

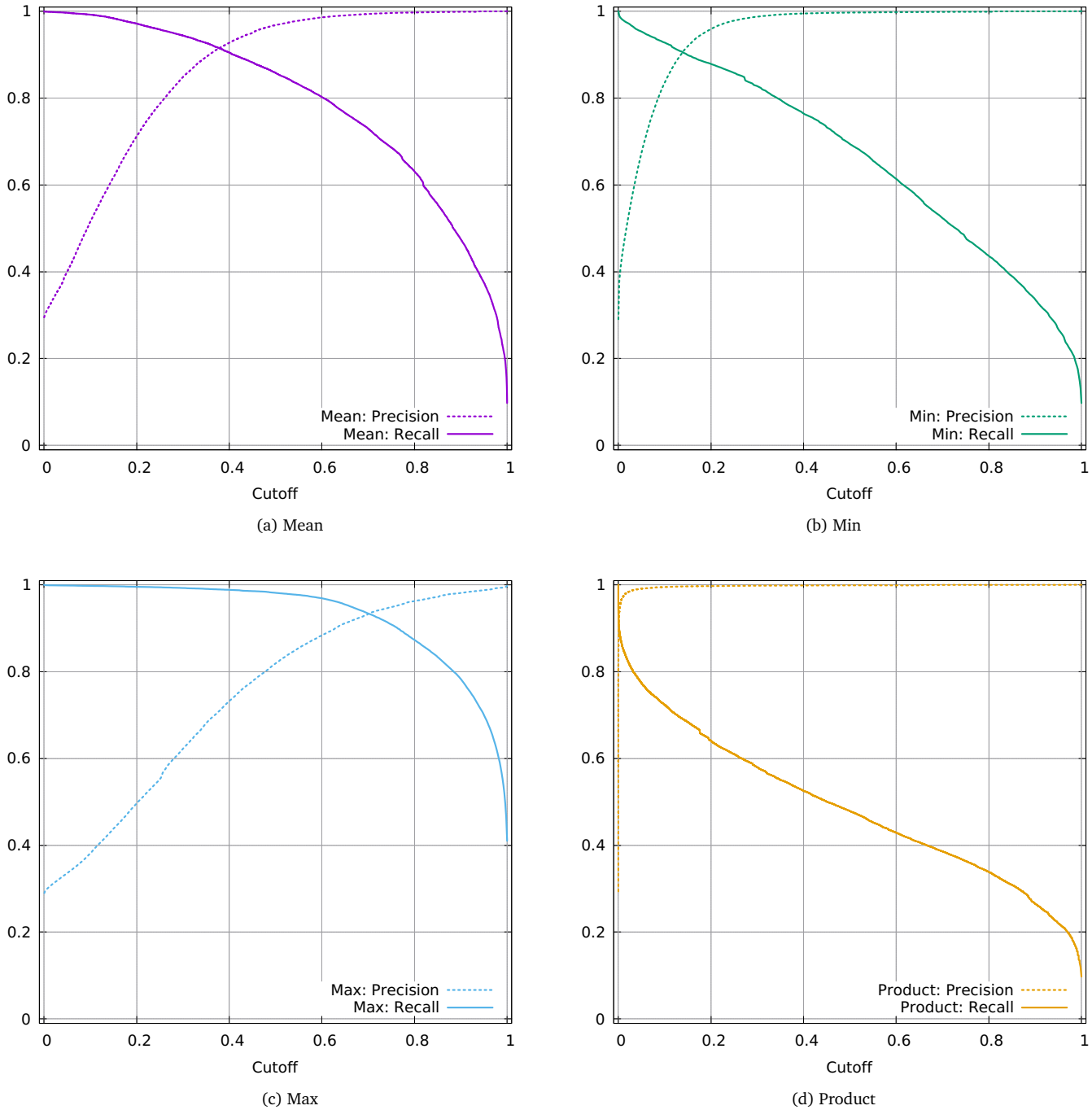
Combining method	AUC	Best F-Measure (@Cutoff)	Best Precision for $Recall > 0.5$	Best Precision for $Recall > 0.99$	Best Recall for $Precision > 0.99$
Mean	0.984	0.916 (@0.411)	0.999	0.547	0.769
Min	0.972	0.917 (@0.21)	0.999	0.377	0.811
Max	0.989	0.934 (@0.702)	0.995	0.692	0.654
Product	0.977	0.917 (@0.003)	0.999	0.388	0.783

We note that, *intuitively* the `min` and `max` combining methods are expected to behave in ways that can be simply expressed. `Min` should result in a *safe* classifier because only apps that are predicted as very likely to be a malware by **all** underlying mono-classifiers will end-up with a high final outcome.

By contrast, `Max` shoud result in a more *aggressive* malware detector, since any app predicted as very likely to be a malware by **at least** one underlying mono-classifier will be attributed a high outcome.

These intuitions are confirmed by Figure 9.6, as we can notice that for `Min`, the Precision is higher than the Recall for the vast majority of possible Cutoff values, while for `Max`, the Recall is above the Precision for the majority of Cutoff values.

Furthermore, Figure 9.6 (d) shows that using `Product` creates a malware detector that is even *safer* than `min`. Indeed, when multiplying values, the impact of one or several very low values is more important than with `Min`, thus only apps for which the consensus amongst underlying mono-classifiers is strong are attributed a high score.



(a) Mean

(b) Min

(c) Max

(d) Product

Figure 9.6: Evolution of Precision and Recall against Cutoff for different combining methods

9.5 Conclusion

In this chapter, we demonstrated that our feature set based on Code Metrics is suitable for Malware Detection. We built a classifier trained on one month worth of data and tested it on the following month. We then combined several classifiers, each trained on one month, and empirically verified a trend: The more one-month-classifiers combined, the higher the performance metrics.

We then investigated what are the characteristics of malware detectors built with different methods of combining classifiers. We showed that while the ROC curves of those multi-classifiers have similar shapes and their AUC are not significantly different, Malware Detectors built with different methods of combining underlying classifiers exhibit totally different characteristics.

ROC Curves are a precious tool to determine the range of possible values for the couple (Recall, False Positive Rate). ROC curves however are unable to help a practitioner choose a Cutoff value.

Whether the primary goal of an experimenter is a high Precision or a high Recall, Figure 9.6 can provide useful insights and help choose both the combining method and the Cutoff value that best fit the practitioner's objectives.

Furthermore, using multi-classifiers create virtually no increase of the classification time because it is trivial to run in parallel many classifiers on a test set.

Chapter 10

Towards a Practical Malware Detector

Contents

While we introduced in the previous chapter families of multi-classifiers that exhibit high performance on a real *in-the-wild* setting, the problem of Android Malware Detection with Machine-Learning Techniques is not a solved issue.

In this chapter, we present several research directions that, we envision, would benefit the malware detection community.

10.1 Predicting Predictors

One of the issues associated with building a practical machine learning-based malware detector for Android applications is to acquire an idea of how dependable the resulting malware detector is.

We believe that not being able to know how well a malware detector will behave on an unknown dataset is one of the most important reasons why machine learning-based malware detectors—despite the vast amount of scholarly literature devoted to the topic—seem to be left mostly unused by the practitioners who would need them the most i.e., Android application markets' owners.

Our work conducted in chapter 9 provide insights on the general shape of Precision and Recall curves against Cutoff, helping to choose both combining method and a Cutoff value. However, many more studies would be necessary to verify if our findings hold for any feature sets.

Also, performing the same experiments than in chapter 9 for a long period of time would increase the confidence in our findings. For example, empirically showing, across one year of *in-the-wild* data, that a malware detector built by combining mono-classifiers trained on the six previous months always lead to similar performance metrics and to similar Precision and Recall values for several given Cutoff values would provide a reasonable way to predict the performance of a malware detector in a *realistic situation*.

By *realistic situation*, we mean a situation where it is not possible to compare the output of a malware detector to a reference classification. Indeed, the very purpose of a malware detector is to detect malware in absence of an existing ground truth.

It is therefore important to devise methodologies that allow practitioners to estimate the behaviour of a malware detector in such a situation or, in other words, to *predict* the reliability of a malware detector.

10.2 Beyond Detection: Explanation

Building a Malware Detector that outputs either a boolean class (GOODWARE / MALWARE) or a real number—hopefully—strongly correlated to the likelihood of being a malware is a hard task. We believe nonetheless that the malware detection community needs to aim higher.

There is only so much trust that can be placed in a black box supposedly able to detect malware, but unable to *convince* us.

Most machine learning-based malware detectors presented in literature are black boxes—including those presented in this thesis. Recently, researchers introduced malware detection approaches like Drebin (Arp et al., 2014) that not only tell malware and goodware apart, but also provide elements of explanation regarding why one given application is reported as a malware.

Similarly, when a machine-learning algorithm manages to learn a model of what discriminates goodware from malware, it is nearly impossible for the experimenter to gain knowledge from that model. In general, the knowledge discovered by the algorithm cannot be translated in a set of meaningful statements about what is a malware and what is a goodware.

10.3 Divide and Conquer

The idea of splitting a hard problem into many easier sub-problems could be applied to the malware detection problem with—we believe—great benefits.

Maybe trying to build one model that would be efficient at detecting malware across the entire space of possible applications is an unreasonably optimistic goal.

The malware detection problem offers several potential ways of being split. In chapter 9, we investigated how splitting the malware detection problem on the time dimension could help build better classifiers.

Another possible dimension for dividing the problem would be to group applications by *families* and try to independently model each *family*. Many possibilities exist regarding how such families could be defined. Apps could be grouped according to their size, their use of any given library or of specific API, the permissions they request or use, the quantity of multimedia assets they embark, etc.

It is conceivable that a) high precision classifiers could be built for several of such groups, and b) collectively, those high precision classifiers cover the entire application space.

Furthermore, splitting simultaneously on several dimensions could result in each application being part of several groups, enabling the use of multi-classification.

10.4 Feature Sets

Finding better feature sets is, and probably always will be, a cornerstone of all approaches based on Machine-Learning. The rapid evolution of both goodware and malware makes it unlikely for one given feature set to perform well for a long period of time.

In particular, it is now common for Android applications to download parts of their code at runtime. Therefore, only using static analysis, as we did in this thesis, means we are blind to such codes. We think improving the existing dynamic analysis tools to acquire and analyse dynamically-loaded code would also improve current malware detection approaches.

Chapter 11

Conclusion

Contents

In this thesis, we have pursued the goal of building dependable malware detectors for Android applications using Machine-Learning Techniques.

We demonstrated several methodology shortcomings in state-of-the-art approaches, and we investigated possible paths to improve large-scale, reliable and practical malware detectors.

11.1 Dataset

As discussed in chapter 4, we built a large dataset of Android applications, that we shared with the research community.

Two reasons were key for our decision to commit to building and managing such a collection of application despite the non-trivial amount of efforts and IT equipment this endeavour requires.

First, we considered the usual size of datasets used in the literature to be too small to conduct experiments whose results would be significant and generalisable.

Secondly, when every single approach is tested on its own non-public dataset, it is nearly impossible to reproduce scientific results, and it makes the standard academic task of comparing a new approach to existing ones more akin to clairvoyance than to science.

The dataset we collected was, and still is used by research teams in France, in Germany and in Singapore, for software engineering and for security research. It is, to the best of our knowledge, by far the biggest Android application dataset available to the research community.

Also, such a large dataset enabled us to perform a survey of the malware landscape of Android applications. This clear picture we obtained can bring objective data to mitigate the scary reports by antivirus vendors or articles on security incidents often found in mainstream media as well as the reassuring discourses of devices manufacturers and of Google.

11.2 Evaluation of Malware Detectors

An important part of this thesis is dedicated to what we called “The Art of Evaluating Malware Detectors”. Indeed, fully understanding the meaning of any malware detection experiment results is very hard, and requires a lot of practice.

After reading any academic paper on Android malware detection, a non-expert would probably come to the conclusion that this is a solved problem.

The sheer number of new malware and the time it takes to detect those are powerful hints that the Android malware detection problem is not solved yet.

Furthermore, as we discussed in depth, several biases may interfere with the generalisability of experimental results. While not insurmountable *per se*, these biases—if not taken into account—have important consequences when moving malware detection approaches from the lab to the wild.

11.3 Towards a Dependable Malware Detector for Android

We proposed in this thesis possible solutions to make machine learning-based Android malware detectors more dependable and thus more likely to be used outside the field of academic research.

By carefully studying the biases and shortcomings of evaluation processes, we were able to craft malware detectors whose performance in the wild are not only good, but can also be—to some extent—estimated *a priori*. We believe this last point has been one of the factors explaining why machine learning based malware detectors are seemingly so rarely used by those who would nonetheless need them. Practitioners have to know how reliable an approach is before it is deployed in real-world situations.

Bibliography

Aafer, Y., Du, W., & Yin, H. (2013). Droidapiminer: mining api-level features for robust malware detection in android. In *Proceedings of the international conference on security and privacy in communication networks*. SecureComm.

Allix, K., Bissyandé, T. F., Jerome, Q., Klein, J., State, R., & Le Traon, Y. (2014). Empirical assessment of machine learning-based malware detectors for android: measuring the gap between in-the-lab and in-the-wild validation scenarios. *Empirical Software Engineering*, 1–29. doi:10.1007/s10664-014-9352-6

Allix, K., Bissyandé, T. F., Jérome, Q., Klein, J., State, R., & Le Traon, Y. (2014). Large-scale machine learning-based malware detection: confronting the "10-fold cross validation"scheme with reality. In *Proceedings of the fifth acm conference on data and application security and privacy* (pp. 163–166). CODASPY '14. San Antonio, Texas, USA: ACM. doi:10.1145/2557547.2557587

Allix, K., Bissyandé, T., Klein, J., & Le Traon, Y. (2015). Are your training datasets yet relevant? an investigation into the importance of timeline in machine learning-based malware detection. In F. Piessens, J. Caballero, & N. Bielova (Eds.), *Engineering secure software and systems* (Vol. 8978, pp. 51–67). Lecture Notes in Computer Science. Springer International Publishing. doi:10.1007/978-3-319-15618-7_5

Allix, K., Jérome, Q., Bissyandé, T. F., Klein, J., State, R., & Le Traon, Y. (2014). A forensic analysis of android malware: how is malware written and how it could be detected? In *Computer software and applications conference (compsac)*.

Alpaydin, E. (2010). *Introduction to machine learning* (2nd). The MIT Press.

Amos, B., Turner, H., & White, J. (2013). Applying machine learning classifiers to dynamic android malware detection at scale. In *Wireless communications and mobile computing conference (iwcmc), 2013 9th international* (pp. 1666–1671). doi:10.1109/IWCMC.2013.6583806

AndroGuard. (2013). Apktool for reverse engineering android applications. Accessed: 2013-09-09. Retrieved from https://code.google.com/p/androguard/

AppBrain. (2013a). Comparison of free and paid android apps. Accessed: 2013-09-09. Retrieved from http://www.appbrain.com/stats/free-and-paid-android-applications

AppBrain. (2013b). Number of available android applications. Accessed: 2013-09-09. Retrieved from http://www.appbrain.com/stats/number-of-android-apps

Apvrille, A. & Strazzere, T. (2012, May). Reducing the window of opportunity for android malware gotta catch 'em all. *Journal of Computer Virology*, *8*(1-2).

Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., & Rieck, K. (2014). Drebin: effective and explainable detection of android malware in your pocket. In *Proceedings of the network and distributed system security symposium*. NDSS.

Arzt, S., Rasthofer, S., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., ... McDaniel, P. (2014). Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Conference on programming language design and implementation (pldi)*.

Azzini, A., De Felice, M., & Tettamanzi, A. (2012). A comparison between nature-inspired and machine learning approaches to detecting trend reversals in financial time series. In A. Brabazon, M. O'Neill, & D. Maringer (Eds.), *Natural computing in computational finance* (Vol. 380, pp. 39–59). Studies in Computational Intelligence. Springer Berlin Heidelberg. doi:10.1007/978-3-642-23336-4_3

Barrera, D., Kayacik, H., van Oorschot, P., & Somayaji, A. (2010). A methodology for empirical analysis of permission-based security models and its applications to android. In *Proceedings of acm conference on computer and communications security*. CCS.

Bartel, A., Klein, J., Monperrus, M., Allix, K., & Le Traon, Y. (2012, May). *Improving privacy on android smartphones through in-vivo bytecode instrumentation*.

Bartel, A., Klein, J., Monperrus, M., & Le Traon, Y. (2012, June). Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot. In *Acm sigplan workshop on the state of the art in java program analysis (soap)*. Beijing, China.

Bickford, J., O'Hare, R., Baliga, A., Ganapathy, V., & Iftode, L. (2010). Rootkits on smart phones: attacks, implications and opportunities. In *Hotmobile '10*. Maryland.

Bissyandé, T. F., Thung, F., Wang, S., Lo, D., Jiang, L., & Réveillère, L. (2013, March). Empirical Evaluation of Bug Linking. In *17th European Conference on Software Maintenance and Reengineering (CSMR 2013)* (pp. 1–10). Genova, Italy. Retrieved from http://hal.archives-ouvertes.fr/hal-00807272

Böhme, R. & Moore, T. (2012). *Challenges in empirical security research*. Singapoore Management University.

Boshmaf, Y., Ripeanu, M., Beznosov, K., Zeeuwen, K., Cornell, D., & Samosseiko, D. (2012, August). Augur: aiding malware detection using large-scale machine learning. In *Proceedings of the 21st usenix security symposium (poster session)*.

Bowes, D., Hall, T., & Gray, D. (2014, April). Dconfusion: a technique to allow cross study performance evaluation of fault prediction studies. *Automated Software Engg. 21*(2), 287–313. doi:10.1007/s10515-013-0129-8

Breiman, L. (2001). Random forests. *Machine learning*, *45*(1), 5–32.

Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., & Sadeghi, A.-R. (2011, April). *Xmandroid: a new android evolution to mitigate privilege escalation attacks* (Technical Report No. TR-2011-04). Technische Universität Darmstadt.

Burguera, I., Zurutuza, U., & Nadjm-Tehrani, S. (2011). Crowdroid: behavior-based malware detection system for android. In *Spsm '11* (pp. 15–26). Chicago, Illinois, USA. doi:10.1145/2046614.2046619

Canfora, G., Mercaldo, F., & Visaggio, C. A. (2013). A classifier of malicious android applications. In *Availability, reliability and security (ares), 2013 eight international conference on*.

Cesare, S. & Xiang, Y. (2010). Classification of malware using structured control flow. In *Proceedings of the eighth australasian symposium on parallel and distributed computing - volume 107* (pp. 61–70). AusPDC '10. Brisbane, Australia: Australian Computer Society, Inc.

Chakradeo, S., Reaves, B., Traynor, P., & Enck, W. (2013). Mast: triage for market-scale mobile malware analysis. In *Proceedings of acm conference on security and privacy in wireless and mobile networks*. WISEC.

Chan, P. P., Hui, L. C., & Yiu, S. M. (2012). Droidchecker: analyzing android applications for capability leak. In *Wisec '12* (pp. 125–136). Tucson, Arizona, USA: ACM. doi:10.1145/2185448.2185466

Chau, D. H., Nachenberg, C., Wilhelm, J., Wright, A., & Faloutsos, C. (2010). Polonium: tera-scale graph mining for malware detection. In *Acm sigkdd conference on knowledge discovery and data mining*.

Cococcioni, M., Corucci, L., Masini, A., & Nardelli, F. (2012). Svme: an ensemble of support vector machines for detecting oil spills from full resolution modis images. *Ocean Dynamics*, *62*(3), 449–467. doi:10.1007/s10236-011-0510-8

Cohen, W. W. (1995). Fast effective rule induction. In *Machine learning-international workshop then conference* (pp. 115–123). Morgan Kaufmann Publishers, Inc.

Cortes, C. & Vapnik, V. (1995). Support-vector networks. *Machine Learning*, *20*(3), 273–297. doi:10.1007/BF00994018

Davi, L., Dmitrienko, A., Sadeghi, A.-R., & Winandy, M. (2011). Privilege escalation attacks on android. In *Isc'10* (pp. 346–360). Boca Raton, FL, USA: Springer-Verlag.

Demme, J., Maycock, M., Schmitz, J., Tang, A., Waksman, A., Sethumadhavan, S., & Stolfo, S. (2013). On the feasibility of online malware detection with performance counters. In *Proceedings of the 40th annual international symposium on computer architecture* (pp. 559–570). ISCA '13. Tel-Aviv, Israel: ACM. doi:10.1145/2485922.2485970

Desnos, A. (2012). Android: static analysis using similarity distance. In *Hicss '12* (pp. 5394–5403). Washington, DC, USA: IEEE Computer Society. doi:10.1109/HICSS.2012.114

Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., & Sheth, A. N. (2010). Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Osdi'10* (pp. 1–6). Vancouver, BC, Canada: USENIX Association.

Enck, W., Octeau, D., McDaniel, P., & Chaudhuri, S. (2011). A study of android application security. In *Proceedings of the 20th usenix conference on security* (pp. 21–21). SEC'11. San Francisco, CA: USENIX Association.

Enck, W., Ongtang, M., & McDaniel, P. (2009). On lightweight mobile phone application certification. In *Ccs '09* (pp. 235–245). Chicago, Illinois, USA: ACM. doi:10.1145/1653662.1653691

Felt, A. P., Finifter, M., Chin, E., Hanna, S., & Wagner, D. (2011). A survey of mobile malware in the wild. In *Spsm '11* (pp. 3–14). Chicago, Illinois, USA: ACM. doi:10.1145/2046614.2046618

Gascon, H., Yamaguchi, F., Arp, D., & Rieck, K. (2013). Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 acm workshop on artificial intelligence and security*. AISec.

Google. (2012, February). Android and security (bouncer announcement). http://googlemobile.blogspot.fr/2012/02/android-and-security.html. Accessed: 2014-06-14.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009, November). The weka data mining software: an update. *SIGKDD Explor. Newsl. 11*(1), 10–18. doi:10.1145/1656274.1656278

He, H. & Garcia, E. (2009). Learning from imbalanced data. *Knowledge and Data Engineering, IEEE Transactions on*, *21*(9), 1263–1284. doi:10.1109/TKDE.2008.239

Henchiri, O. & Japkowicz, N. (2006). A feature selection and evaluation scheme for computer virus detection. In *Proceedings of the sixth international conference on data mining* (pp. 891–895). ICDM '06. Washington, DC, USA: IEEE Computer Society. doi:10.1109/ICDM.2006.4

Höbarth, S. & Mayrhofer, R. (2011, June). A framework for on-device privilege escalation exploit execution on android. In *Proc. IWSSI/SPMU*. San Francisco, CA, USA.

Hornyack, P., Han, S., Jung, J., Schechter, S., & Wetherall, D. (2011). These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th acm conference on computer and communications security* (pp. 639–652). CCS '11. Chicago, Illinois, USA: ACM. doi:10.1145/2046707.2046780

Hutchins, M., Foster, H., Goradia, T., & Ostrand, T. (1994). Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on software engineering* (pp. 191–200). ICSE.

Idika, N. & Mathur, A. P. (2007, February). *A survey of malware detection techniques*. Purdue University.

ITU. (2008, November). *Information technology – Open Systems Interconnection – The Directory: Public-key and attribute certificate frameworks Technical Corrigendum 2*. ITU. ITU-T Recommendation X.509. Genebra.

Jacob, A. & Gokhale, M. (2007). Language classification using n-grams accelerated by fpga-based bloom filters. In *Proceedings of the 1st international workshop on high-performance reconfigurable computing technology and applications: held in conjunction with sc07* (pp. 31–37). HPRCTA '07. Reno, Nevada, USA.

Jerome, Q., Allix, K., State, R., & Engel, T. (2014, June). Using opcode-sequences to detect malicious android applications. In *Communications (icc), 2014 ieee international conference on* (pp. 914–919). doi:10.1109/ICC.2014.6883436

Jones, J. A. & Harrold, M. J. (2005). Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th ieee/acm international conference on automated software engineering* (pp. 273–282). ASE. ACM.

Kephart, J. O. (1994). A biologically inspired immune system for computers. In *In artificial life iv: proceedings of the fourth international workshop on the synthesis and simulation of living systems* (pp. 130–139). MIT Press.

Kolter, J. Z. & Maloof, M. A. (2006, December). Learning to detect and classify malicious executables in the wild. *J. Mach. Learn. Res. 7*, 2721–2744. Retrieved from http://dl.acm.org/citation.cfm?id=1248547.1248646

Kubat, M., Holte, R., & Matwin, S. (1998). Machine learning for the detection of oil spills in satellite radar images. *Machine Learning*, *30*(2-3), 195–215. doi:10.1023/A:1007452223027

Liu, M. & Lu, J. (2014). Support vector machine – an alternative to artificial neuron network for water quality forecasting in an agricultural nonpoint source polluted river? *Environmental Science and Pollution Research*, *21*(18), 11036–11053. doi:10.1007/s11356-014-3046-x

Liu, X. & Liu, J. (2014, April). A two-layered permission-based android malware detection scheme. In *Mobile cloud computing, services, and engineering (mobilecloud), 2014 2nd ieee international conference on* (pp. 142–148). doi:10.1109/MobileCloud.2014.22

Madabhushi, A., Shi, J., Feldman, M., Rosen, M., & Tomaszewski, J. (2006). Comparing ensembles of learners: detecting prostate cancer from high resolution mri. In R. Beichel & M. Sonka (Eds.), *Computer vision approaches to medical image analysis* (Vol. 4241, pp. 25–36). Lecture Notes in Computer Science. Springer Berlin Heidelberg. doi:10.1007/11889762_3

Mann, H. B. & Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, *18*(1), 50–60.

McLachlan, G., Do, K.-A., & Ambroise, C. (2005). *Analyzing microarray gene expression data*. Wiley. com.

Nauman, M., Khan, S., & Zhang, X. (2010). Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Asiaccs '10* (pp. 328–332). doi:10.1145/1755688.1755732

Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., & Le Traon, Y. (2013). Effective inter-component communication mapping in android with epicc: an essential step towards holistic security analysis. In *Proceedings of the 22nd usenix security symposium*.

Peng, H., Gates, C. S., Sarma, B. P., Li, N., Qi, Y., Potharaju, R., ... Molloy, I. (2012). Using probabilistic generative models for ranking risks of android apps. In *Proceedings of acm conference on computer and communications security*. CCS.

Perdisci, R., Lanzi, A., & Lee, W. (2008a). Mcboost: boosting scalability in malware collection and analysis using statistical classification of executables. In *Proceedings of the annual computer security applications conference*. ACSAC.

Perdisci, R., Lanzi, A., & Lee, W. (2008b). Classification of packed executables for accurate computer virus detection. *Pattern Recognition Letters*, *29*(14), 1941–1946. doi:10.1016/j.patrec.2008.06.016

Pieterse, H. & Olivier, M. (2012). Android botnets on the rise: trends and characteristics. In *Proceedings of the conference on information security for south africa* (pp. 1–5). ISSA.

Pouik & G0rfi3ld. (2012, April). Similarities for fun & profit. *Phrack*, *14*(68). Retrieved from http://www.phrack.org/issues.html?id=15&issue=68

Quinlan, J. R. (1993). *C4. 5: programs for machine learning*. Morgan kaufmann.

Rossow, C., Dietrich, C., Grier, C., Kreibich, C., Paxson, V., Pohlmann, N., ... van Steen, M. (2012, May). Prudent practices for designing malware experiments: status quo and outlook. In *Security and privacy (sp), 2012 ieee symposium on* (pp. 65–79). doi:10.1109/SP.2012.14

Sahs, J. & Khan, L. (2012). A machine learning approach to android malware detection. In *Intelligence and security informatics conference (eisic), 2012 european* (pp. 141–147). IEEE. doi:10.1109/EISIC.2012.34

Santos, I., Penya, Y. K., Devesa, J., & Bringas, P. G. (2009). N-grams-based file signatures for malware detection. In *Iceis* (pp. 317–320).

Schultz, M., Eskin, E., Zadok, E., & Stolfo, S. (2001). Data mining methods for detection of new malicious executables. In *Security and privacy, 2001. s p 2001. proceedings. 2001 ieee symposium on* (pp. 38–49). doi:10.1109/SECPRI.2001.924286

Su, X., Chuah, M., & Tan, G. (2012). Smartphone dual defense protection framework: detecting malicious applications in android markets. In *Eighth ieee international conference on mobile ad-hoc and sensor networks*. MSN.

Tahan, G., Rokach, L., & Shahar, Y. (2012, June). Mal-id: automatic malware detection using common segment analysis and meta-features. *J. Mach. Learn. Res. 98888*, 949–979.

Van Hulse, J., Khoshgoftaar, T. M., & Napolitano, A. (2007). Experimental perspectives on learning from imbalanced data. In *Proceedings of the 24th international conference on machine learning* (pp. 935–942). ICML '07. Corvalis, Oregon: ACM. doi:10.1145/1273496.1273614

Vidas, T. & Christin, N. (2013). Sweetening android lemon markets: measuring and combating malware in application marketplaces. In *Codaspy '13*.

Viennot, N., Garcia, E., & Nieh, J. (2014). A measurement study of google play. In *The 2014 acm international conference on measurement and modeling of computer systems* (pp. 221–233). SIGMETRICS '14. Austin, Texas, USA: ACM. doi:10.1145/2591971.2592003

Visaggio, C. A., Pagin, G. A., & Canfora, G. (2013). An empirical study of metric-based methods to detect obfuscated code. *International Journal of Security & Its Applications*, *7*(2).

Vural, E., Çetin, M., Erçil, A., Littlewort, G., Bartlett, M., & Movellan, J. (2009). Machine learning systems for detecting driver drowsiness. In K. Takeda, H. Erdogan, J. Hansen, & H. Abut (Eds.), *In-vehicle corpus and signal processing for driver behavior* (pp. 97–110). Springer US. doi:10.1007/978-0-387-79582-9_8

Wu, D.-J., Mao, C.-H., Wei, T.-E., Lee, H.-M., & Wu, K.-P. (2012). Droidmat: android malware detection through manifest and api calls tracing. In *Information security (asia jcis), 2012 seventh asia joint conference on* (pp. 62–69). doi:10.1109/AsiaJCIS.2012.18

Yerima, S., Sezer, S., McWilliams, G., & Muttik, I. (2013). A new android malware detection approach using bayesian classification. In *Advanced information networking and applications (aina), 2013 ieee 27th international conference on* (pp. 121–128). doi:10.1109/AINA.2013.88

Zhang, B., Yin, J., Hao, J., Zhang, D., & Wang, S. (2007). Malicious codes detection based on ensemble learning. In *Proceedings of the 4th international conference on autonomic and trusted computing*. ATC. Hong Kong, China.

Zhou, W., Zhou, Y., Jiang, X., & Ning, P. (2012). Detecting repackaged smartphone applications in third-party android marketplaces. In *Codaspy '12* (pp. 317–326). ACM.

Zhou, Y., Wang, Z., Zhou, W., & Jiang, X. (2012). Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In *Ndss'12*.

Zhou, Y. & Jiang, X. (2012). Dissecting android malware: characterization and evolution. In *Proceedings of the 2012 ieee symposium on security and privacy* (pp. 95–109). SP '12. Washington, DC, USA: IEEE Computer Society. doi:10.1109/SP.2012.16

Zhou, Y., Zhang, X., Jiang, X., & Freeh, V. W. (2011). Taming information-stealing smartphone applications (on android). In *Trust'11* (pp. 93–107). Pittsburgh, PA: Springer-Verlag.